

IIS Security and Programming Countermeasures

By Jason Coombs (jasonc@science.org)

Introduction

This is a book about how to secure Microsoft Internet Information Services for administrators and programmers whose work includes a requirement for information security, a computer industry specialty field commonly referred to as infosec. In this book the terms information security and infosec are used interchangeably with the more friendly term data security. This is not a book about hacking, cracking, and the tools and techniques of the bad guys, the so-called black hat hackers. This book teaches computer professionals and infosec specialists how to build secure solutions using IIS. It is your duty to secure and defend networked information systems for the benefit of the good guys who are your end users, clients, or less technical coworkers.

There is nothing you can do that will transform a programmable computer running Microsoft Windows from its vulnerable condition to an invulnerable one. Every general purpose programmable computer is inherently vulnerable because it is controlled by software and is designed to allow new software to be installed or executed arbitrarily. Network computing based on programmable general purpose computers will never be safe from an information security perspective. Eliminating the feature of general purpose programmability from a networked computer and replacing its software with firmware reduces but does not eliminate vulnerabilities. These are immutable realities of present day computing and, as always, reality represents your biggest challenge. Microsoft is in business to get as much of your money as possible using whatever means will work at a given moment and in this respect they know virtually no equal in the software business.

Unfortunately, Microsoft truly does not care about security. You will see why in this book. To Microsoft, your possession of a microprocessor turns you into a customer, a source of potential profit. Just as your possession of a pair of eyeballs turns you into a potential customer for media giants who would sooner see you put in prison for violating arbitrary intellectual property laws than thank you sincerely for the money you've paid to them over the years, Microsoft will never do anything (willingly) that reduces its competitive position by reducing its access to your microprocessors or relinquishing some of its leverage over you. Never mind that these same corporations and media giants are responsible for laws such as the Digital Millennium Copyright Act (DMCA), that you may one day find yourself accused of violating based on something a microprocessor appears to have done over which you allegedly had control or authority, because of political contributions and special-interest lobbyist politics.

Giving you real control over your computers is not in the best interest of capitalism nor law enforcement because such control would reduce profits and hinder prosecutions. If you don't think these issues are a part of the complex and volatile modern world of data security then you'll be surprised how a little knowledge of the subject will change your perceptions. Just remember that if it becomes difficult for Microsoft to execute code on your computers, it will become more difficult for them to extract money from your bank accounts by selling you more software. The business methods used by Microsoft have so badly compromised the safety of those who use Microsoft software that I've called publicly for Microsoft to give away, free of charge to all existing Microsoft customers, the

latest build of Windows code that incorporates, for the first time, security remediations produced as a result of the Trustworthy Computing Initiative. These architectural security fixes are not just new features that you might like to have and may choose to pay for, they are the first attempt Microsoft has ever made to create a product that is safe to use and free from severe defects. That Microsoft has failed yet again to achieve a reasonable level of safety for its products will become apparent in the coming months, but this does not change the fact that Microsoft profited enormously by selling severely defective products in the past and owes a debt of apology to every person and business that has been harmed by their actions and inactions. We'll never see this apology in the real world, of course, just as we may never see Microsoft software that incorporates common sense security countermeasures. It is just not in Microsoft's best-interest to do what is in your best-interest, and this alone should cause you serious concern. Many other businesses draw a line that they choose not to cross, out of respect and care for their customers and for the benefit of the general public. Microsoft draws no such line, and it has thus become a despicable company run by despicable people.

Information security is a constant process that never achieves its objective. Given this fact, many people choose to do something else with their time and money. People who faint at the sight of blood make poor surgeons. Likewise, computer programmers or administrators who insist that computers are inherently trustworthy under certain circumstances make terrible information security professionals. Before you can expect any computer system to be trustworthy you must abandon any hope you might have that a technological solution may exist and reconsider the criteria by which you judge a computer to be trustworthy. Technically, a computer can be considered trustworthy if it is provably under your exclusive control, performing only operations that are known or expected, and you are certain to detect any behavior or condition that would indicate otherwise. Your risk exposure to a computer can be considered reasonable if the computer is trustworthy and you are aware of, and prepared to respond to, incidents that may occur due to malfunction or malfeasance.

Unlike a guide to black hat hacker mischief, and more applicable to daily programming or administration tasks than a guide for so-called white hat hackers who conduct penetration tests and employ hacking tools for the purpose of ensuring data security or discovering new vulnerabilities before malicious hackers do, IIS Security shows you where threats exist in data networks and information systems built around Microsoft IIS to enable threat comprehension but makes no effort to give detailed instructions on perpetrating exploits. There is plenty to read on the Internet on that subject. This book shows how to harden IIS and its hosted Web applications and services against attacks so that all known, and hopefully all possible, black hat exploits can be prevented with solid data security technology, secure Web application code, application-specific threat countermeasures, and a security policy appropriate to the level of protection required for each server box.

IIS Security assumes that you are using IIS version 4, 5, 5.01, or 6.0 with an emphasis on versions 5 and 6. IIS versions 5 and 5.01 are only available for Windows 2000 or Windows XP Professional and IIS 6 only available in the Windows .NET Server OS family. Although some of the instructions in this book pertain specifically to one version of IIS and therefore imply use of a particular OS (NT 4, Win2k, XP or .NET Server) you will find the majority of the instructions relevant to your environment because Windows XP Professional and .NET Server share a common code base derived from Windows 2000

which in turn derives from Windows NT 4. Most of the Windows 2000-specific instructions also apply to members of the Windows .NET Server OS family if only in a historical context. You may not need to follow all of the instructions in this book if you are using Windows .NET Server and IIS 6 because much of the best practices knowledge, registry settings, and secure configuration options have been preconfigured for IIS 6 in the most secure locked-down setting. Consider historical Windows 2000 instructions to be informative if you need to lower security settings on your Windows .NET Server with IIS 6 rather than increase them.

If you are still using a Windows NT 4 system with IIS 4, which include NT 4 derivatives such as BackOffice Server 4.5, you would realize a performance benefit for IIS if you were to upgrade, but IIS version 4 can be made as secure as newer versions and Microsoft does not plan to stop supporting version 4 nor stop releasing security hotfixes for it. If you do use IIS 4, the critical thing to understand is that unlike IIS 5 and 6, IIS 4 must not be used to host applications on behalf of multiple unrelated parties. IIS 4 can only be adequately secured for application hosting if the applications belong to a single organization. To securely host Web sites for multiple organizations under IIS 4 you must disable all dynamic content and application services, such as ISAPI extensions, and serve only static HTML content. Removing all active content and applications to achieve security is a drastic measure, and while this book can be used as a guide to dismantling programmability interfaces exposed by IIS, it assumes that your objective is to build, deploy, and manage secure applications around IIS not revert to serving static text and pictures. The bottom line is that if you host applications for other people, upgrade to Windows 2000 or Windows Server 2003.

The sample code shown in this book, and there is no shortage of it, is not meant solely for computer professionals whose business cards bear the title of Programmer. Administrators will find the code useable as a basis for administrative scripting, and the code samples are tailored with this in mind. The boundaries between administrator and programmer have blurred recently with the release of new versions of Windows, the .NET Framework, and new programming platforms such as Windows Script Host that appeal to and solve problems for anyone who manages, uses, or programs Windows-based computers. This is, in many ways, a continuation of the trend toward more manageable computing and it places an emphasis on skills rather than job titles. Programmers tend to crossover into administrative realms and administrators tend to crossover into programming realms with greater frequency.

The succinct and pretentious goal of the code samples shown in this book is to reveal for each topic the essential lines of code that you should have been able to find published prominently in the documentation. In most cases you will need the documentation to make full use of the code, and possibly even to understand it depending on how much knowledge of the .NET Framework and C# you currently possess. Reproducing in print that which is better suited to online distribution makes little sense, and copying and pasting technical documentation makes for a mostly-useless book that insults your intelligence. Therefore the chapters in this book make no attempt to be comprehensive with respect to programming instructions or the more mundane technical background that you can easily acquire yourself and probably already have if you're bothering to read this in the first place.

Information Security Threats

Natural Disasters and Disaster Recovery

Well Known Security Holes

Disabling IIS Request Processing During Installation

Modifying iis.inf on a File Share or Custom Installation CD

Permissions Problems

Security Blind Spots

Buffer Overflow Vulnerabilities

Malicious Software

Hacking Tools

Password Crackers

Port Scanners

Worms and Viruses

Trojan Horse Programs

Web Crawlers and Search Engines

Software Bugs

Network Security Holes

Man in The Middle Vulnerabilities

DNS Hijacking and Spoofing

Proxy Farms and Cache

Privacy Leaks

The Human Threat

Script Kiddies

Rogue Employees and Sabotage

Eavesdropping

Unnatural Disasters

Chapter 2: Microsoft Internet Information Services 43

Architecture Design

The IIS Platform

inetinfo.exe

Metabase

IIS Admin Service

Internet Server Application Programming Interface (ISAPI)

WAM Director

Microsoft Transaction Server (mtx.exe)

COM+ Surrogate Out of Process Host (dllhost.exe)

IIS 6 Isolation Mode Worker Process (w3wp.exe)

A Brief History of inetinfo.exe

IIS Version 4.0

IIS Version 5.0

IIS Version 6.0

Kernel Mode Request Queueing with http.sys Duct Tape

Multiple Worker Processes

Process Isolation Mode

Web Administration Service

Integration with Windows Server OS

Win32 and inetinfo.exe

Auditing inetinfo.exe

Programmable Extensibility
Active Template Library ISAPI Extensions and Filters
Integration with .NET
ASP.NET
.NET Passport
Integration with COM+
Nuclear and Extended Family
Nuclear SMTP, NNTP, FTP, and Publishing Point Services
Products That Extend The IIS Foundation

Chapter 3: Server Farms 70

Understanding Farms and Gardens
Load Balancing Proxies and Switches
Windows Network Load Balancing
LAN/WAN Security and Firewalls
Intrusion Detection Systems
Client Proxy Farms
Dynamic Traffic Filtering by IP Address
Detecting Scans, Probes, and Brute Force Attacks
Deploying Honeypots to Detect Intruders
Cryptology and Intrusion Countermeasures
Digital Signatures and Credential Trust
Asymmetric Cryptography for Bulk Encryption
Creating Public Key Cryptostreams
Decrypting Public Key Cryptostreams

Chapter 4: Platform Security	99
Security Foundations of Windows	
Windows Registry and NTFS	
Primary and Impersonation Access Tokens	
Security Identifiers	
Security Descriptors, DACLs, SACLs, and ACEs	
Component Object Model	
Disabling Distributed COM	
Application ID, Class ID, and Program ID	
Type Libraries	
Interface Hardening in Microsoft Networks	
Multihomed Windows Networking	
Interface Linkage and Service Bindings	
Port Filtering and Network Service Hardening	
Monitoring SYN/ACK TCP Packets on Open and Filtered Ports	
RFC 1918 Route Hardening	
Windows Application Installation and Execution	
Windows File Protection	
Windows File Checker and Signature Verification Utilities	
Chapter 5: ASP.NET Security Architecture	143
Built-in Security Primitives	
Authentication	
Authorization	
Session Management	
Impersonation	

Encryption

Code Access Security

Internet Information Services ASP.NET Host

ASP.NET Script Engine

ASPNET_ISAPI.DLL File Types

Windows .NET/2000 Server OS or Windows XP Professional

System.Security

System.Security.Principal.IIdentity

System.Security.Principal.GenericIdentity

System.Security.Principal.IPrincipal

System.Security.Principal.GenericPrincipal

Custom Dynamic Role Membership with a Custom IPrincipal Class

System.Web.Security

Web Application Security

Understanding Insecure Authenticated Sessions

Chapter 6: ASP.NET Application Security..... 160

Using ASP.NET Authentication

System.Web.Security.WindowsAuthenticationModule

System.Security.Principal.WindowsIdentity

System.Security.Principal.WindowsPrincipal

.NET Passport

Forms Authentication

Authorization of Request Permissions

File Authorization

URL Authorization

Custom Authorization with Application Code

ASP.NET Impersonation No-Code Authorization

Fundamentals of ASP.NET Data Encryption

System.Security.Cryptography

Encrypting Data

Decrypting Data

Generating and Managing Encryption Keys

Verifying Data Integrity Using Hash Codes

Chapter 7: Secure Scripting for Web Applications 188

Know Your Audience/Enemy

Securing Anonymous Session State

Meeting and Greeting Anonymous Users

JavaScript Enabled Clients Can Help to Secure Anonymous Sessions

Prove-You're-Human Countermeasures

Garbage In Garbage Out

Principle of Least Astonishment

Cross Site Scripting

XSS in Services Running Alongside IIS

XSS in Active Server Pages Script

Automatic Launch Vectors for XSS

Input Validation Rules

Assume Purposeful Web Client Tampering

Harden Session Identifiers

Impose Explicit Character Set Encoding

Safe Scripting for Internet Applications	
HTTP and HTML Cache-Busting Techniques	
URL-Encoded Name/Value Pair Disclosure	
Forcing SSL Access To Scripts	
Chapter 8: TCP/IP Network Vulnerabilities	214
Tracing The Flow of Trust in TCP/IP Networks	
Domain Name System Vulnerabilities	
Presumed Trust in Unauthenticated Connections	
Source Address Spoofing and Routing Restrictions	
Developing Trustworthy Clients	
DNS Spoofing and Hijacking Attacks	
Vulnerabilities Produced by Insecure DNS	
Preventing Hijacking Attacks	
Detecting and Preventing Man in the Middle Attacks	
Automated Detection of Proxies and a Deny-First Countermeasure	
DNS Pooling as an Automated Countermeasure Against DNS Attacks	
Preventing Denial of Service Attacks	
Protecting IIS with a Restrictive Security Policy	
Chapter 9: Transaction Processing Security	236
Windows On-Line Transaction Processing	
Automated Transactions	
Distributed Transaction Coordinator Lockdown	
Crash Recovery of Pending Transactions	
Hardening Business Processes	

Preserving Common Sense Protections	
Creating Transactionless Idempotent Operations	
Hardware Foundations of Secure OLTP	
Write-Ahead Logging and Shadow Blocks	
Transaction Logging to Facilitate Decontamination	
Replicating Logs with Point-to-Multipoint Communications	
Tracking Responsible Security Principals	
Logging The Call Stack	
Hashing The Call Stack	
Using a Call Stack Hash as a Secret Key	
Transactional Message Queues	
Microsoft Message Queue Service	
Using a Response Queue with a Reverse Proxy Server	
Chapter 10: Code Safety Assurance.....	263
Countermanding Untrustworthy Code	
Profiling and Blocking Application Initialization	
Restricting NTFS ACL Execute Permissions	
Compiler Security Optimizations	
Modeling Vulnerable Code	
Exception Handler Function Pointer Replacement	
Suppressing Bad Memories	
Analyzing Win32 API Dependencies	
Chapter 11: ISAPI Health and Hardening	285
Fundamentals of Reliable Extensions or Filters	
Malformed Requests with Malicious Payloads	

Careful ISAPI Development and Deployment

MFC Parse Maps

Custom ISAPIs Without MFC

Global Interceptors

Request Chaining and Rewriting

Layered Security Wrapping of ISAPIs

Managed Healthcare for ISAPI Extensions

Worker Process Recycling

Worker Process Kernel-Mode Overlapped I/O

Browser Client Context ISAPI W3Who.dll

Common Gateway Interface ISAPI Alternatives

Complete Process Isolation with WSH CGI

Application Extension Mapping for The WSH Script Engine

HKEY_USERS\DEFAULT Hive for IUSR_MachineName with WSH

Chapter 12: Authentication Credentials 306

HTTP Challenge/Response WWW-Authenticate

Preventing Automatic Logon Attempts via HTTP

Unencrypted Basic Authentication with Base64 Encoding

Forcing Browser Clients to Reauthenticate

Basic Authentication Support in .NET Framework Classes

Sending an Authorization Header in HTTP Requests with .NET

Protection Realms and Credential Cache

Basic Authentication Support in Windows HTTP Services (WinHTTP)

Server-Side User Token Cache Settings

Microsoft Windows Digest Authentication	
Integrated Windows Authentication	
Windows NT LAN Manager (NTLM) Version 2 Authentication	
Active Directory and Kerberos Authentication	
Anonymous Authentication	
ASP.NET Forms Authentication	
Anatomy of a Forms Authentication Ticket	
Dynamic Modification of Credentials in web.config	
Chapter 13: Trojans and Stealth Rootkits	331
Blended Threat Characteristics	
Concept Worms and Viruses	
Zero-Day Exploits and Custom Trojan Code	
Spyware and Other Trojans Associated with Clients	
Vendor Relations and Infection Prevention	
Hotfix Digital Signatures and Authentic Code Verification	
CryptCATAdminCalcHashFromFileHandle and SipHashData	
SHA-1 File Hashing with .NET Framework Cryptography	
Positive Proof of Signature Verification Failure	
Assume Your Vendor is Incompetent	
Windows XP/.NET Software Restriction Policies	
Adding Yourself as The Sole Trusted Software Publisher	
Restricting Software Execution Based on Hash Codes	
Chapter 14: Certificates for Encryption and Trust	361
Large-Scale Dangers of Certificate Chain Trust	
Choose A Public Key To Trust And Stick To It	

Security Alerts Don't Resolve Certificate Chain Vulnerability
Managing Certificates and Certificate Stores
Removing Default Root Certificates From Each Store
Using The MMC Certificates Console Snap-In
Configuring Trusted Certificates for IISAdmin and W3SVC Services
Issuing Certificates with Windows Certificate Services
Producing Certificate Signing Requests with Web Enrollment Support
Sending Certificate Signing Requests with The CCertRequest Client
Generating Your Own Code Signing Key Pairs and Certificates
Trust Only Your Own Digital Signatures and Root CAs
Signing and Timestamping Trusted Windows Binaries or Custom Code
Windows File Protection Arbitrary Certificate Chain Vulnerability
Windows File Protection Old Security Catalog Vulnerability
Creating and Signing Your Own Security Catalog Files
Designing a Multi-Level Certification Hierarchy
Client Certificate Authentication
Issuing Client Certificates with Web Enrollment
Mapping Client Certificates to User Accounts
Many-To-One Mappings
One-To-One Mappings
Secure Sockets Layer Encryption

Chapter 15: Publishing Points..... 391

File Transfer Protocol Service

IIS 6 FTP User Isolation Mode

Using Hidden FTP Virtual Directories
WebDAV Distributed Authoring and Versioning
Disabling WebDAV for Your Entire IIS Box
FrontPage Server Extensions (FPSE)
Publishing Point IIS Admin Object ADSI Scripts
Automating Web Site Creation
Scripting New Web Site Creation
Configuring Metabase Permissions
Configuring Site Member and Site Developer User Authentication
Configuring Membership Authentication
Editing Metabase Classes and Properties
FTP Service Publishing Point Provisioning with ADSI

Chapter 16: Proving Baseline Security 415

Microsoft Baseline Security Analyzer
Well-Known Vulnerability Scanning with MBSA
Analyzing The XML Source Data for Baseline Security
Network Security Hotfix Checker (Formerly HFNETCHK)
Hash Verification with Checksum Correlation

Chapter 1 Web Threats

When you undertake to secure IIS from threats and detect vulnerabilities in applications hosted under IIS, you enter the realm of the information security profession, known as infosec. Infosec is a specialty in the computer field that seeks to analyze, mitigate, and respond to threats to information systems. The importance of infosec is in the hard-won insight and awareness of risk it brings to everyday computing. To understand what infosec is and how it applies to the development and deployment of secure information systems built around Microsoft IIS it helps to understand the nature of threats to computer security. A computer security threat is anything or anyone that can cause a computer to do things you don't want it to do or disrupt its normal operation. In general, the CPU in a programmable computer is only supposed to execute machine code instructions that facilitate useful and productive data processing and communication that fits with the purpose and intent of the information system as it was originally deployed. Any deviation from this purpose, whether caused by unforeseen unintended consequences including software bugs or caused by the actions of a malicious attacker, represents a potential vulnerability. This includes a deviation from normal operation that causes the information system to stop fulfilling its intended purpose, a so-called Denial of Service (DoS) condition.

In addition to the focus on eliminating threats and ensuring health and normal operation of information systems, infosec is also concerned with risk management, privacy, protection of trade secrets and preservation of evidence that may be of forensic value in the event of legal problems. It includes topics such as cryptography, transaction management, data loss and theft prevention, online and offline data storage media integrity and mean time to failure analysis, development and deployment of intrusion detection systems, security incident response and forensic analysis of malicious code and compromised systems. Information warfare, penetration tactics and defense countermeasures, disaster recovery, access control systems, biometric identification technology and associated databases, content filtering, virus and Trojan detection, embedded systems programming for security applications, smart cards, digital surveillance, digital signatures, key escrow, and key certification authority trust management systems are all part of the infosec field. Any or all of these topics may be important to you as you build and operate secure networked information systems using a Windows Server operating system and Internet Information Services. The chapters that follow cover most of these topics so that the overall level of security you achieve building and deploying custom applications using Microsoft server software on a TCP/IP network can meet industry best practices and guarantee, at a minimum, that no security breach or incident will go unnoticed on your IIS boxes.

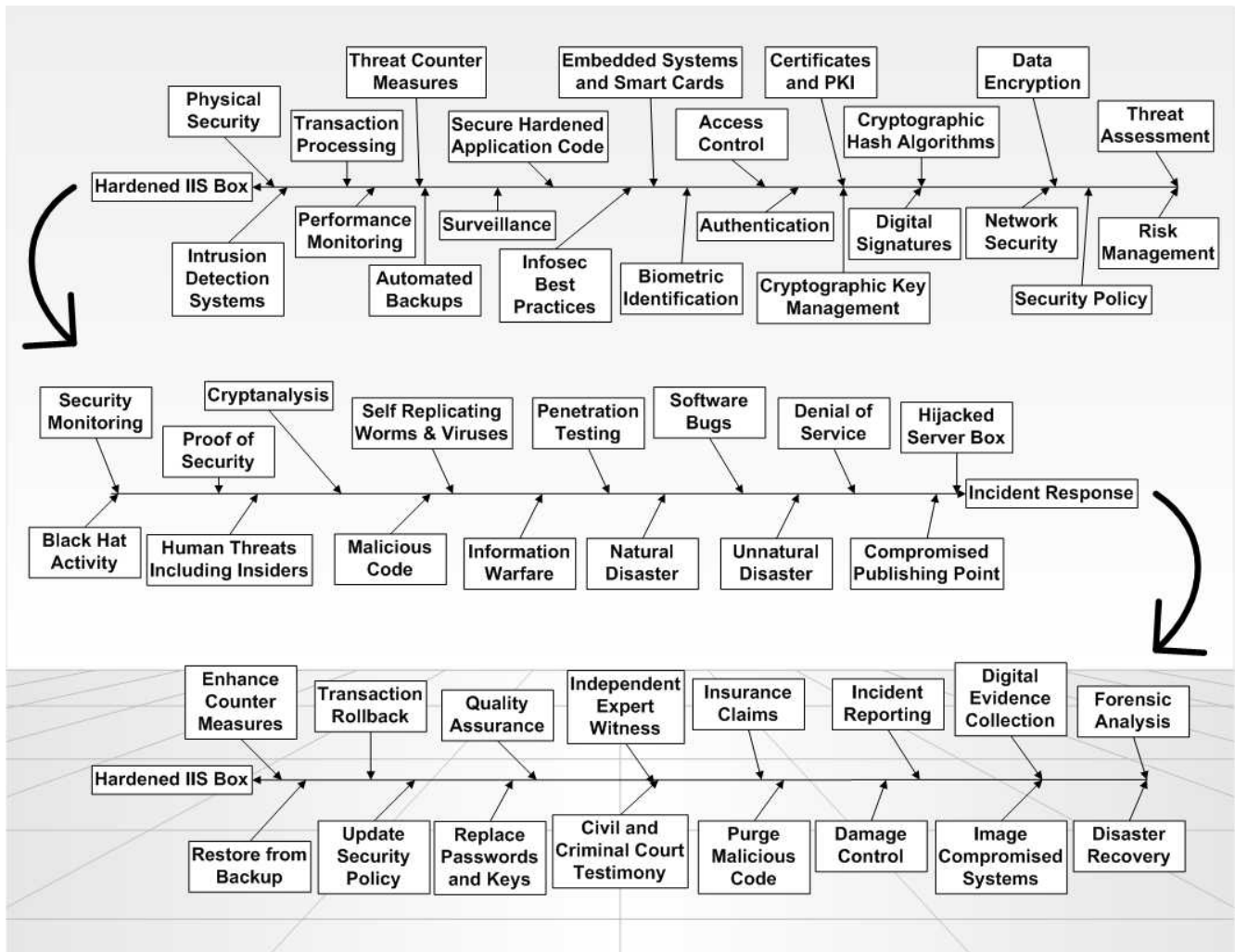


Figure 1-1: Infosec and IIS Incident Response

Figure 1 shows the infosec defense-incident-defense continuum. To the extent that a compromised IIS box stands alone and has no privileges on nor special access to other systems and networks, incident response may end once IIS has been returned to a hardened state. Otherwise incident response includes all potentially affected hardware and software. This suggests that small redundant isolated server groups offer the most practical threat containment and most efficient incident response. Like a fire break used to protect structures from a raging wildfire, boxes that you can figuratively burn during incident response to quickly eliminate the potential of undiscovered threats offer something of practical value to data security. Setting up decoy boxes called honeypots to attract malicious attacks like flies to honey may also give advance threat warning by sacrificing essentially disposable computers and data.

Information security concepts and techniques are applied for two reasons. The first reason is self-evident: to protect computer systems and data from unauthorized tampering or access and make sure they are able to serve users and fulfill automated duties without interruption. Every user, administrator, executive, and decision-maker who relies on or manages information systems needs some level of assurance of data security and system integrity. The second reason is computer forensics. As an infosec specialty, computer forensics analyzes and documents information systems and explains them

forensically to assist with a civil or criminal legal proceeding or to prove compliance with privacy and other laws. Part of incident response, aside from restoring compromised or affected systems to an original secure state, is the application of computer forensics tools and techniques to gather reliable and accurate digital evidence to present to other people who need to know about the security incident. Digital evidence comes in many forms and may be communicated to others for many reasons, including presentation through legal process in civil or criminal court, reports given to corporate management, security administration efforts with Internet service providers, and accounting for financial damages caused by computer security incidents.

The World Wide Web, or your organization's intranet Web, is possibly the most inherently vulnerable of all information systems. In addition to conventional security dependencies on network routers, hubs, proxies, firewalls, and physical wiring and facilities, the Web and its variations are exposed to attacks and disruptions beyond those that impact other client/server software systems because of the Web's design. The fact that the Web gives control of user interface look and feel to Web page authors and, by design, caters to so-called thin clients (which in practice are thin veneers applied to thick clients) makes Web applications more difficult to secure than conventional software. The difficulty stems from the complexity of interactions between client and server, programmer and administrator, network engineer and third party service provider, thin client program and its thick operating system host, not to mention malicious hacker and victim. Thin clients are unable to defend themselves against attacks the way that thick clients can, as you have no control over the features and functionality of the software client or its platform. This places nearly the entire security burden on the Web server. The more potential the Web brings with its extensibility, universal accessibility, and programmability, the more difficult it becomes to manage the security of everything connected to it.

For instance, end users who might otherwise be able to detect security problems with the software they use because it suddenly changes its behavior are conditioned to have no expectations other than to accept whatever they see as the intended behavior of a Web application. And also by design, the Web seeks to eliminate the user's ability to distinguish between software running on their local computer, which might access or destroy data that is important to them, and software running on the server. The fact that the same information technology that is used to serve data and applications on the global public Internet is deployed throughout organizations' own private data networks, and even on end users' client computers where there may be no reason for network services to exist in the first place, means that black hat hackers can fine-tune their malicious skills in the relative safety of the anonymous public Internet and then mount attacks against your private network. Or, worse yet, they can come to work for your company and mount attacks from inside.

Data Security Threats to Web Servers

You are the first threat to data security that you should fully comprehend. Anything that you can do to a server box running IIS anyone else can also do if they can impersonate you successfully. Every shortcut you find to make your life simpler, such as granting yourself remote access ability to administer server boxes from home, leaving out passwords for systems that can only be accessed from inside your private network, or deploying a single sign on solution to grant yourself access to all password-protected resources

through a single authentication step, makes protection of your authentication credentials more critical. The risk is not limited to human attackers or even to programs that might steal your password and use it to login with your privileges. If you only need to authenticate once when you login, malicious code that you execute unknowingly or that can force itself to execute by exploiting buffer overflow vulnerabilities in services you launch may end up running in your security context with your privileges. Without additional safeguards, such malicious code doesn't even have to worry about stealing passwords, it can simply install a Trojan that keeps the malicious code running or launches it again next time you login.

Rather than tearing down walls to simplify management, you should erect more of them to prevent yourself from carrying out administrative and programming tasks without additional unique authentication for each type of activity. One time use passwords at each point of authentication combined with password lists or password generation algorithms to determine the password to use to satisfy the next authentication challenge provide one of the only means of protection against password theft from keystroke loggers, shoulder surfing, hidden cameras, or other tactics used by intruders to capture secret passwords as they are entered at the keyboard. Figure 1-2 illustrates a layered approach to authentication with predefined password lists. Such lists are known as "one-time pads" if they are used as encryption keys rather than as access control system passwords. Each layer in the figure represents some distinct activity, it doesn't matter what except that the activity layers should make sense to you and effectively compartmentalize damage that may be done by an intruder who manages to get through one or two layers but fails to get through all of them. In order to compromise one time pad authentication entire lists of passwords must be intercepted rather than a single password because passwords become invalid once they are used. To further confuse an attacker who does intercept the one time pads but fails to eavesdrop on any successful authentication session the passwords can be encrypted using some simple memorized key that never gets stored with the password lists. This way the password lists themselves can't be used for authentication without additional knowledge of the secret encryption key. One time use password lists, especially when used in a layered authentication model so that a single password doesn't grant unrestricted control of a protected resource, provide substantially better password security at minimal cost and without unreasonable inconvenience to you, the administrator or programmer, or to your end users.

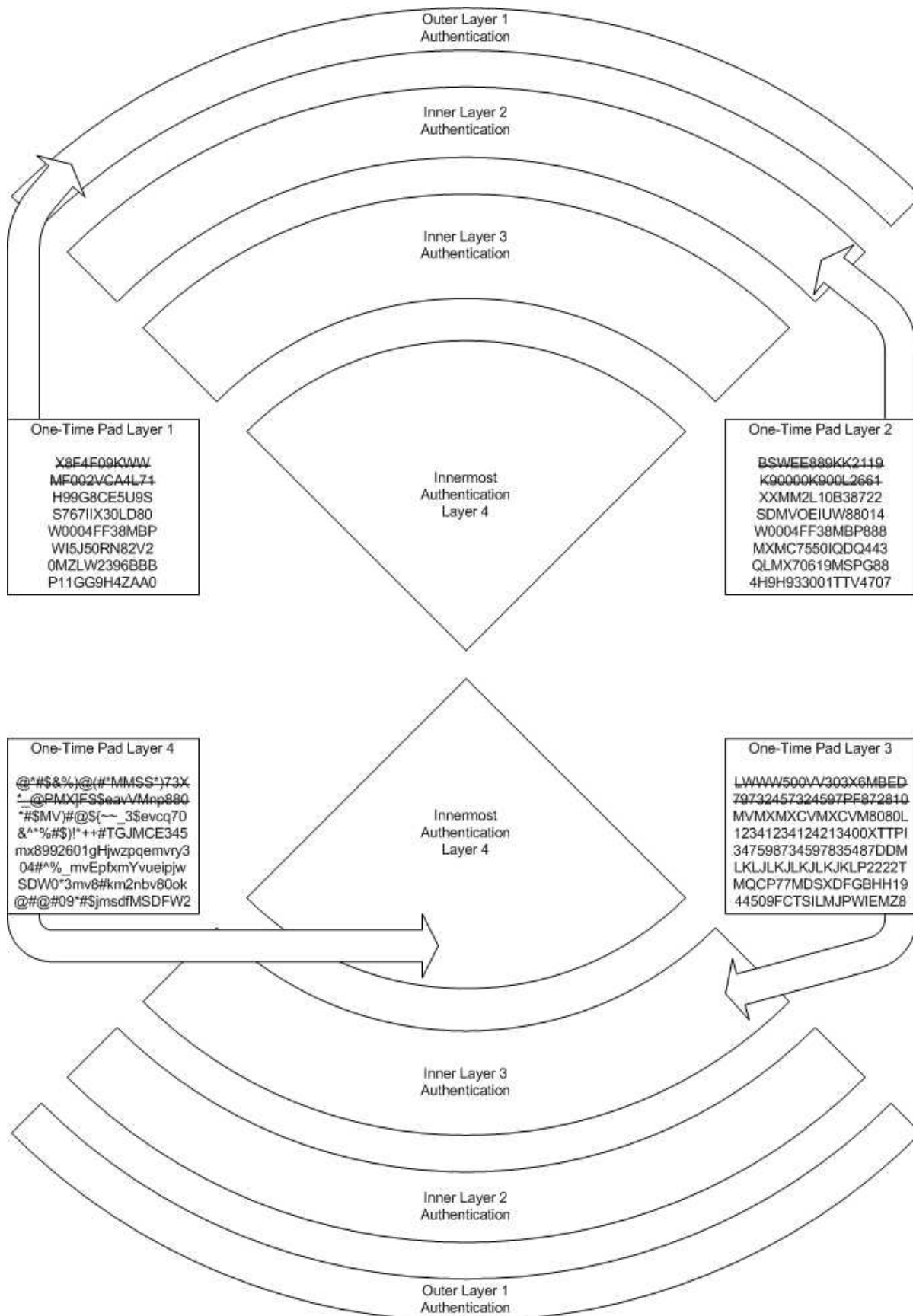


Figure 1-2: Layered Authentication with One Time Use Password Lists

Biometric identification systems and other technology can be used with passwords to enhance authentication, but bear in mind that none of the data security protections you implement do much good when a malicious third party can unplug your computers and carry them out the door then read all the data off the hard drives. The fact that you can connect a hard drive from a crashed system to a freshly-built server box, boot, and access the data on the hard drive may be helpful for disaster recovery, but it should also be viewed as a data security vulnerability. If you're not going to deploy filesystem encryption, it could be pointless to go overboard with complicated, costly, and inconvenient authentication technologies and additional barriers to prevent you from seamlessly and efficiently carrying out your daily administrative or programming duties.

Another subtle threat that it's important to be aware of is the risk that your Web server will be used far more than you expected. Excessive usage can cause Denial of Service (DoS) conditions as network bandwidth becomes saturated or limited server computing resources such as disk space or CPU speed are depleted. It can also lead directly to financial loss in certain situations, such as when your ISP has a billing policy that calculates your bill based on total network traffic or if your Web application provides a live link to a system that charges transaction fees. It's a commonly-known credit card fraud exploit amongst black hat hackers that e-commerce Web sites which perform live credit card verification in order to give customers real-time feedback about the success or failure of their purchase attempt can be used as a credit card number and expiration date oracle. An oracle, in infosec terms, is any system that will give authentic responses to input and thereby enable an attacker to choose invalid inputs repeatedly until the oracle responds with an answer that indicates to the attacker that the input, or a portion thereof, may not be invalid.

Attackers have been known to send thousands of invalid credit card numbers and expiration dates to Web sites looking for those few combinations that result in a successful purchase, and thus reveal a valid credit card number. This type of abuse turns a feature that some Web developers view as essential for every e-commerce business into a financial liability and a disruptive threat to anyone whose credit card number might be discovered through such an oracle attack. A security researcher several years ago discovered an oracle attack against the Secure Sockets Layer (SSL) protocol that used an SSL server's response to about a million carefully constructed invalid messages to correctly deduce the server's secret key. This type of vulnerability places not only future communications at risk, since the attacker could subsequently impersonate the authentic SSL server as well as decrypt any subsequent communication that the authentic server has with clients, but it also places at risk every secure communication the server has ever had in the past using that key because anyone who has been carefully recording all traffic to and from the server can retroactively decipher the encrypted portions of all previous traffic. Be aware of this sort of risk, and change your encryption keys as often as possible to minimize the damage that is done when one of your keys is compromised.

Usage floods don't always have a malicious origin. The popular Web site Slashdot.org noticed that other Web sites to which Slashdot.org would hyperlink often suffered DoS conditions due to the flood of visitors who would simultaneously converge on the sites. Thus a DoS outage caused by a usage flood without malicious intent is sometimes referred to as the Slashdot effect. The average quality of visitors during a sudden usage flood often decreases as curiosity seekers or misdirected users who have no real interest

in returning to a Web site at a later time clog up the works. There are many reasons that a usage flood may occur, and DoS conditions will continue to be a problem whenever demand exceeds the computing and network resources needed to service the demand.

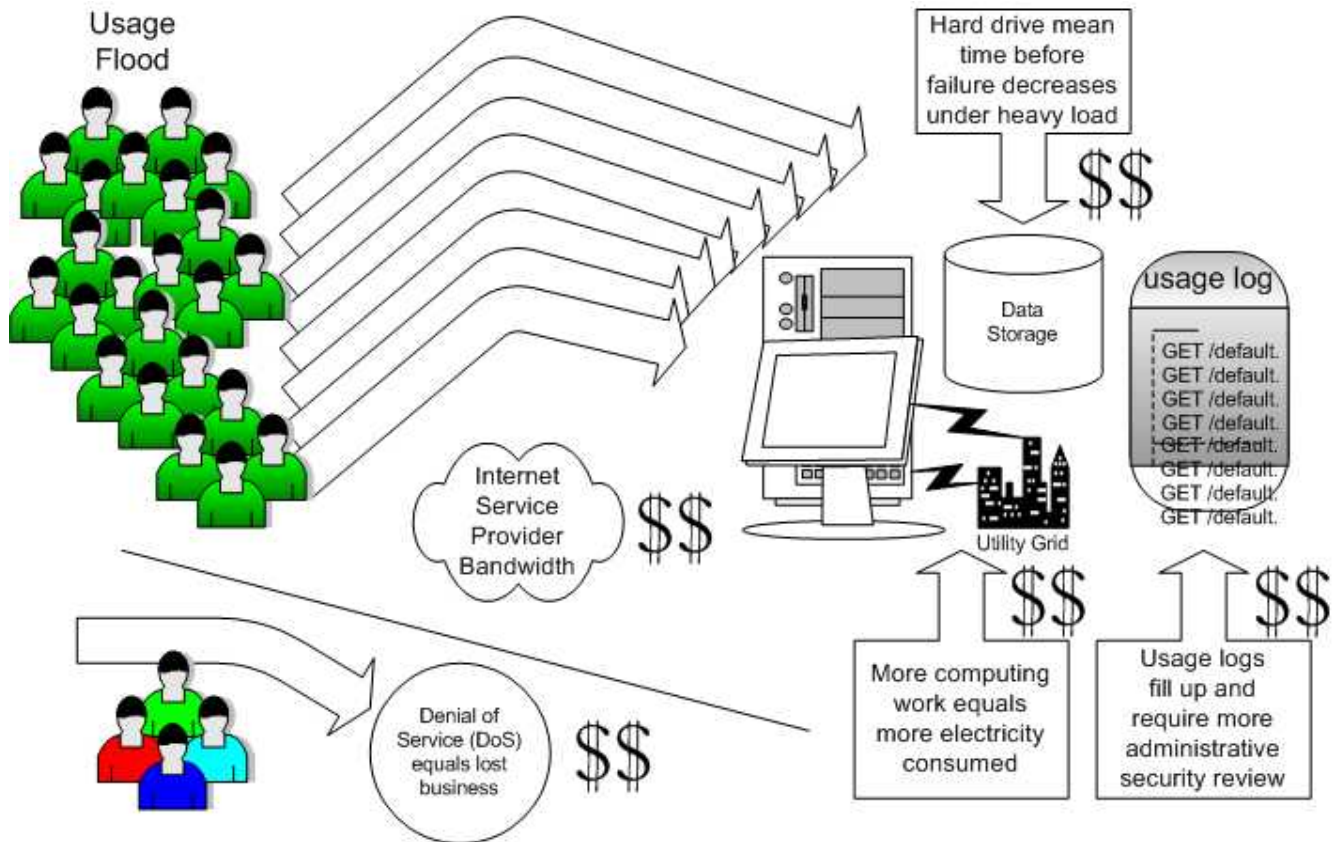


Figure 1-3: A usage flood can cause damage without malicious intent

Excessive usage can lead indirectly to financial loss, too, as hard drives and other equipment wear out and must be replaced sooner than expected. Figure 1-3 depicts the typical financial impact of a usage flood. There is often a direct correlation between amount of usage and time and effort required to manage security, review daily logs, and perform routine administration tasks. If you're not prepared to spend this extra time, it may be better to leave your servers offline or configure rate limits so that the rate of usage can't exceed your ability to manage usage growth while ensuring security and protecting against threats. Unless you're certain that a large amount of traffic is expected and acceptable, every spike in traffic should be noted and investigated, and malicious sources of traffic spikes should be carefully investigated and potentially blocked from future access to your servers.

Natural Disasters and Disaster Recovery

It may seem somewhat obvious to you that a natural disaster is a threat to data security in a tangible, physical security sense. But being prepared for a natural disaster and having an effective disaster recovery plan in place don't stop at drying out flooded data centers, repairing or replacing damaged equipment, and restoring from off-site backups. Think also about where the data came from that you are about to restore. Did it come from an employee's desk at home? If you can't prove that nobody other than the employee had

physical access to the disk from the time the backup was placed there until the time you prepare to restore your systems from it, then it may not be trustworthy. How will you verify that your backup hasn't been tampered with? If you don't have hash codes for your backup stored redundantly in a very safe place, protected differently from if not superior to the protection afforded the backup itself then you may have data but you don't have security.

Hashing algorithms are designed to read data of any length and compute a fixed-length bit sequence that is unlikely to result from hashing other data but that will always result in the same bit sequence from processing the same data. Although it is possible for two different data inputs to hash to the same bit sequence output, it is computationally difficult to discover such circumstances, and it's even more unlikely that the second data set would in any way resemble the first. This means that even if you aren't getting absolute verification of the validity of the input data like you would get comparing every bit of two complete copies of data, you do know that no human or computer is likely to be able to use forged data that hashes to the same hash code as authentic data. The second data set would be essentially random noise, it would not have the structure that the authentic data had. Further, hash codes are cryptographically secure in that the original data can't be discovered, not even a single bit's worth of original data, from cryptanalysis of hash codes. There is much more in this book about hashing and hash algorithms, as hash codes are one of the fundamental tools of data security.

Well Known Security Holes

Many security holes become so widely known that almost nobody is vulnerable to them any longer. In this way even the worst case scenario rampant outbreak of self replicating malicious code burns itself out after a relatively short period of time. Worms and viruses that exploit well known security holes may still propagate, if only because some malicious programmer has intentionally mutated the code in an attempt to create a computer super bug by merging as many replication vectors and vulnerability exploits as possible into something new that gains a foothold in the wild. When well known security holes translate into worms, viruses, and Trojans that are also well known, antivirus software can often detect them reliably. That doesn't mean that you can assume that software publishers, including OS vendors like Microsoft, are always able to prevent retail product distributions from becoming infected with well known malicious code. For another thing, counterfeiting is increasingly common in the software industry, and you may not have received an authentic original retail build from Microsoft inside that shrink-wrapped box you bought at the store. Verifying hash codes of each binary file prior to installation, and running an antivirus scan of the entire CD- or DVD-ROM are critically-important security measures that must be practiced without exception in order to control and prevent the introduction of malicious code along with your installation of retail software.

Programmable computers can be reprogrammed by malicious code so hash code verification of retail software and antivirus scans prior to installation are necessary to guarantee that a hardened system whose security status you are reasonably confident of doesn't fall victim to an attack that can so easily be prevented. It takes a lot more effort to harden a box than it takes to compromise a box through a single careless mistake.

In addition to being a symptom of a poorly-managed network node, hosts that become infected with legacy worms and viruses pose a threat to your servers during initial

installation of unpatched operating systems from the original, authentic, retail distribution media. To protect against infection, which may simply be a nuisance but that could also carry a custom malicious payload of an unusual variety designed to bypass antiviral scanning and take control of your box or leave it vulnerable to later attacks, you can create custom installation media that disables vulnerable services or replaces buggy binaries with versions from service packs or hotfixes. This allows you to get up and running to the point that a proper install of the latest service pack and hotfixes can be performed interactively or through an unattended installation script.

Disabling IIS Request Processing During Installation

During installation of any Windows OS that includes a version of IIS there is normally a period of time after the services have started but before you have applied security patches, hot fixes, and service packs that resolve well known security problems. During this period of time your services are vulnerable to attack, especially if they are running on a box that is connected to a network in order to make installation and configuration easier. You may have a network share from which to install software, you may need access to a domain controller from the system you're configuring, or you may decide it's important or helpful to be connected to a network for some other reason. Obviously it is preferable to leave your system disconnected from the network until after you've finished your installation if possible. When this is not possible or practical there are steps you can take to disable request processing by IIS during installation.

Modifying iis.inf on a File Share or Custom Installation CD

By modifying iis.inf on your installation media you can adjust default settings for IIS to disable anonymous access on the default Web site and take other security precautions during installation. This is easy to do if you are installing from a network share since the installation files are already stored in writeable folders. But for CD-ROM based installation you will need to burn a custom copy of the operating system installation CD. You can make your custom CD bootable just like the original by using a CD burning software package that is compatible with the El Torito standard for ISO-9660 format bootable CDs. Modifying iis.inf prior to installation is only necessary in IIS versions prior to 6.0 as request processing is severely limited by default when you install any member of the Windows .NET Server OS family.

The installation procedure for IIS exists inside a compressed .inf file named iis.in_ on the operating system installation media. The compressed .inf file is located in the same directory as the .cab (cabinet) files that contain groups of related files and programs that may be installed depending upon your configuration selections. To access the iis.inf file for editing you must use the expand.exe utility to uncompress the compressed .in_ file. After expanding iis.in_ open the resulting iis.inf in a text editor and remove each line that appears within the following sections:

```
[ScriptMaps_CleanList]
[InProc_ISAPI_Apps]
[IIS_Filters_SRV]
[IIS_Filters_WKS]
[DefaultLoadFile]
```

```
[START_W3SVC_IF_EXIST]
[START_W3SVC]
[START_MSFTPSVC_IF_EXIST]
[START_MSFTPSVC]
[START_PWS_IF_EXIST]
[START_PWS]
```

Removing these default settings and service startup commands from iis.inf prevents network request processing of active content types and prevents IIS from loading default ISAPI filters or extensions if the services do start for some reason before you've had a chance to install hotfixes and the latest OS service pack. In addition, perform the following steps to configure IISAdmin, W3SVC, and MSFtpsvc services to start manually instead of automatically. To disable these services entirely, simply change the hex value 0x3 at the end of each line to 0x4 to indicate Disabled.

1. Add the following to [register_iis_core_0_values] on a single line:
HKLM,System\CurrentControlSet\Services\IISADMIN,
"Start",0x00010001,0x3
2. Add the following to [register_iis_www_values] on a single line:
HKLM,System\CurrentControlSet\Services\W3SVC,
"Start",0x00010001,0x3
3. Add the following to [register_iis_ftp_values] on a single line:
HKLM,System\CurrentControlSet\Services\MSFtpsvc,
"Start",0x00010001,0x3

Your OS installation will still have IIS but it won't be active for request processing. Because certain other services and configuration steps that require access to the Metabase may launch the IISAdmin service and leave it running, you may still see inetinfo.exe in a list of active processes like the task manager. As you'll find out in Chapter 2, inetinfo.exe represents the IISAdmin service, not request processing capabilities provided by network applications like the IIS World Wide Web Publishing Service. Until additional services are started, inetinfo.exe will not receive and process requests from the network just because it is running due to the IISAdmin service being started.

Permissions Problems

Misconfigured permissions are a constant threat, especially under Windows server operating systems where NTFS Access Control Lists (ACL) can specify different permissions for every user and group. Compared to other server operating systems, which typically allow permissions to be set for only three categories of user; the owner of each file, members of the group associated with each file, and everyone else who belongs to any other group or no group, access controls for files in Windows servers are more configurable. In addition to filesystem permissions, which include data files or programs, Windows servers include the system registry where interfaces and objects are configured and assigned permissions. IIS rely further on the Metabase where hosted sites and Web applications are assigned permissions.

With Windows 2000 and subsequent OS versions there is also active directory which centralizes management of network resources such as user accounts, security policies, and access permissions. Finally, the most recent addition to permissions complexity under Windows, Microsoft's .NET framework for secure managed code offers a new type of permission known as Code Access Security (CAS). With CAS, every managed code object, method, and property can be assigned separate permissions. Invocation of managed code modules registered in the global assembly cache can also be controlled with permissions.

Security Blind Spots

Not looking, or being unable to look, at what's going on in any part of your IIS box creates a security blind spot that compounds the impact of other vulnerabilities and potentially allows exploits and incidents to go unnoticed. An intrusion detection system, or IDS, is a critical component of any secure deployment of IIS. An IDS is responsible for observing all network traffic, and possibly all application usage and machine code execution on your box, to scan for suspicious activity or signs of an attack or vulnerability exploit. Many IIS deployments are protected by firewalls that don't include IDS features, causing critical counter intelligence about attacks in progress to be dropped at the firewall boundary. There is no legitimate reason for users to run port scanners against your IP address range, for example, but in spite of thousands of malicious TCP packets hitting the firewall in addition to the one packet addressed to port 80, firewalls fail to recognize the port scan for what it is and simply pass on the TCP connection request to the firewalled Web server on port 80.

An IDS can automatically block all traffic from a remote address that exhibits suspicious behavior or sends malicious network traffic. It's impossible for a user to accidentally type malicious URLs that are known to trigger buffer overflow exploits, so another IDS function is to pick out such malicious requests and automatically reject them and possibly any other request that originates from the same source address. This brings up a difficult point for optimal security in the real world of the Internet: a single source address can represent thousands or millions of potential end users. Proxy servers are commonplace on the Internet, almost as common as firewalls, and devices that perform Network Address Translation (NAT) such as routers that have embedded firewall features are also very common. NAT devices allow many network nodes to share a single Internet IP address by automatically translating the destination address on each incoming packet to the appropriate LAN address of the node that needs to receive the packet. The NAT device maintains a dynamic table that maps port numbers used for outbound communications on the external network interface with IP address and port number on the internal LAN where a particular node awaits packets addressed to the external interface with a specific port number. Blindly blocking traffic unless it matches a particular port number is better than blindly allowing all traffic to reach your IIS box, but something in between, mediated by an IDS, is a much better security solution.

Buffer Overflow Vulnerabilities

The most important type of vulnerability to protect against is the buffer overflow bug that exists in software that has not been properly security-hardened. Buffer overflows come in several flavors, including the most common stack overflow and the less common heap

overflow. The process by which buffer overflow vulnerabilities are discovered, and the programming flaws that produce them, are well known in the infosec and black hat hacker communities. The question is who has more manpower and resources devoted to discovering buffer overflows, infosec or black hats? Discovery of a novel overflow is the *raison d'être* for black hats, but programmers are typically busy with lots of other things besides putting code through comprehensive forensic analysis procedures looking for potential vulnerabilities. The self evident reality is that black hats will often find vulnerabilities first. One of the only real protections we have is careful security monitoring, outsourced to a managed security monitoring infosec service provider if possible, to capture and analyze attacks as well as successful penetrations that reveal a new vulnerability discovered independently by black hats. Knowledge of the vulnerability can then be disseminated widely to allow everyone to protect themselves before too much damage is done.

Buffer overflow vulnerabilities become well known and well defended faster than most other vulnerabilities because of the extreme threat they represent. This type of vulnerability can allow malicious code to be executed by the attacker. Execution of arbitrary malicious code is one of the worst case scenarios of computer security that requires specific threat reduction countermeasures and unlike subjectively bad things, such as privacy leaks, malicious code execution is considered objectively and universally bad. When a third party gains the ability to execute code on your box without privilege containment, the box no longer belongs to you it belongs to the third party. Even with privilege containment the third party still ends up owning the data and capabilities afforded to the compromised privilege level.

The stack and heap are fundamental programming concepts that you must understand in order to secure any programmable computer from the threat of malicious code. The stack is an area of memory set aside by a compiler for use in managing memory that the compiler knows a program will need in order to execute. The heap is where all other memory used by a program comes from when the program requires more memory at run-time. A program must explicitly request heap memory as it requires additional memory allocations, heap memory isn't automatically managed for a program by the compiler. When source code is converted into object code by a compiler, in order to change human-readable instructions into machine-readable instructions, the compiler knows ahead of time how large to make the stack for each procedure call within the program. Stack memory is defined at compile time for all memory elements that are declared in the source code. This includes all local variables declared in each procedure and the variables created by the compiler to hold the parameters, if any, a function receives when invoked by a caller at run-time. The compiler also adds extra overhead so that it has room on the stack to keep track of things like the location of the stack and where in memory the machine code instruction resides that the microprocessor will be pointed to after a call to a subroutine finishes executing.

Microprocessor design includes support for accessing the stack explicitly, with a push instruction to place bytes onto the stack and a pop instruction to remove bytes from the stack. These instructions, combined with memory registers known as the stack frame pointer and the stack pointer, are used to keep track of where in memory the next bytes will go when more are pushed on the stack and where to begin removing bytes when the processor encounters a pop instruction. A call to a subroutine requires the stack to grow,

which requires a new stack frame so that the called subroutine will have stack space of its own in memory that doesn't conflict with the stack space already in use by other procedures. The compiler can't predict how many subroutine calls there might be as the compiled program code executes, so to keep everything running smoothly the microprocessor pushes the important values from its internal memory registers onto the new stack frame in preparation for each subroutine call. This includes the current instruction pointer; the memory address of the machine code instruction being executed by the microprocessor. Intel processors use the EIP register to store the instruction pointer. It also includes the address of the previous stack frame, which is stored on Intel processors in the EBP register.

Together these two register values saved on the stack enable the called subroutine procedure to return control to the caller. The microprocessor will be instructed at some point to return from the call to the subroutine, at which time it will pop bytes from the stack to restore the previous stack frame to EBP and pop bytes from the stack again to set EIP equal to the stored EIP value. The stack pointer, which uses the ESP register, will end up pointing at the end of the previous stack frame, and anything that gets pushed on the stack from that point forward will overwrite the values stored previously on the stack for use in the subroutine call because those values are no longer needed. The stack is where most buffer overflow vulnerabilities exist due to the relative ease with which the stored EIP value in a new stack frame can be altered. The following Hello World! program illustrates the problem directly.

```
#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
int main(int argc, char* argv[]) {
void * p[2] = {(void *)p[2],(void *)p[3]};
p[2] = (void *)&p[0];
p[3] = (void *)0x00401224;
printf("Hello World!\n");
return 0; }
```

To understand what this Hello World! program does that causes a security problem, you have to examine the stack while the program executes. Figure 1-4 shows the Visual C++ debugger in the midst of inspecting the stack at a breakpoint in program execution just prior to the call to the printf function which will output the Hello World! message. You can see that the local variable, p, which the program declared as an array of two elements of type void * (a generic memory pointer type) begins on the stack at memory address 0x0012ff78 and stretches for 8 bytes (each void * value is 4 bytes in length) to 0x0012ff7f where the hex value of 00 appears just to the left of the hex value 78 in the memory window labeled Address: in Figure 1-4. The first byte of p variable memory can be seen in the figure immediately above the last byte of p's 8-byte memory block, and the first byte contains the hex value C0.

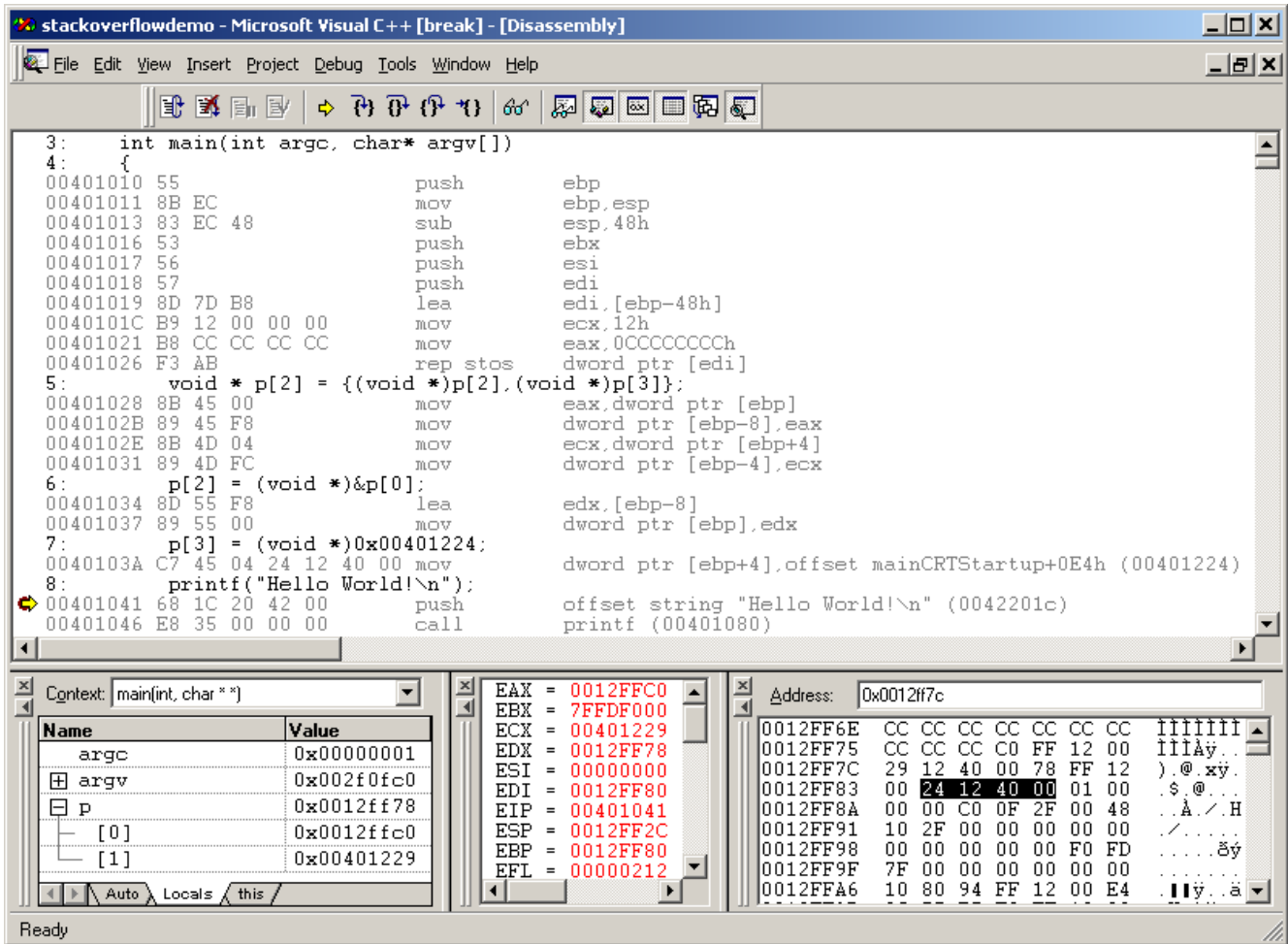


Figure 1-4: Stack buffer overflows are easy to understand

The first thing the Hello World! program does is copy the 4 bytes beginning at memory address 0x0012ff80 to the first element of p and copy the 4 bytes beginning at memory address 0x0012ff84 to the second element of p. It does this by using the array references p[2] and p[3], which both exceed the capacity of the memory buffer set aside for p (an array that only has two elements, p[0] and p[1]) in order to give the microprocessor a few valid “move memory” instructions (mov). As a result of executing these instructions p[0] and p[1], the two elements of the p array, contain the values from memory immediately preceding the variable p on the stack. Note that the stack grows down in memory address as it gets bigger, so the values placed on the stack first are found at higher memory addresses. The new values of p[0] and p[1] are important because they are the authentic previous stack frame address and the authentic calling instruction pointer address, respectively, that were pushed onto the stack when the current stack frame was created by the call to the main procedure. Those values must be left alone in memory or else when main finishes executing, as it will immediately following the printf function call when the return instruction is encountered, the microprocessor will pop the wrong values off the stack and be unable to return to the authentic EIP and EBP register settings in order to continue executing whatever code comes after the call to the main subroutine.

The problem is that this Hello World! program, using the same array references p[2] and p[3] as were just discussed, proceeds to clobber the original authentic values of EBP and EIP

as pushed on the stack. At the memory address of p[2], which contains the previous stack frame address from the EBP register, the program writes the address of p[0]. This will cause the processor to reuse the same stack frame, starting at 0x0012ff80, as is currently being used (notice the current value of EBP in the register list in the bottom middle of Figure 1-4) when the main function finishes and prepares to return. The final step taken by the program is to replace the value of the saved EIP with a value of its own.

This is precisely what malicious code does when it exploits a stack overflow vulnerability, it sets things up in memory so that the stack itself triggers execution of the exploit by passing bad data back to the EIP register when the microprocessor is expecting authentic data. The memory address used by the Hello World! program for its replacement EIP, 0x00401224, happens to be only 5 bytes away from the authentic EIP value of 0x00401229 which you can see in Figure 1-4 stored in the address of p[1] beginning at 0x0012ff7c. The machine code instruction at that address is the original call to the main function that started the program executing in the first place. The buffer overflow exploit shown in the Hello World! program isn't malicious, it simply results in an unusual sort of infinite recursion. Something similar is accomplished by calling main recursively from within the main function, as shown in the following code.

```
#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
int main(int argc, char* argv[]) {
printf("Hello World!\n");
main(0,(char **)NULL);
return 0; }
```

Except that with standard recursion like this you don't end up with an infinite recursive loop. Eventually the stack overflows itself, since it has to grow with each function call in order to keep track of all those stack frames and return EIP addresses. It doesn't matter that the addresses are always the same, they still have to be stored so that when recursion ends (which obviously it never will in the code as shown) things can be unraveled properly and processing can continue. The buffer overflow depicted in Figure 1-4 is the basis of the well-known stack buffer overflow threat, whereas the stack overflow that results from a typical recursive subroutine that never wraps up its recursion is just a bug that results in an error message. Buffer overflow threats in the wild, rather than in canned demonstrations, aren't coded in the manner shown in Figure 1-4 so they can't explicitly reference element three or four in a two-element array in order to get the overflow exploit started. They rely instead on the existence of buffers whose length is unchecked by application code as the application code fills those buffers. All it takes to defend against the stack buffer overflow and harden code that is vulnerable to this type of attack is to strictly enforce fixed-length buffer access. For more on this and related topics for secure software development see *Writing Secure Code* by Michael Howard and David Leblanc, published by Microsoft Press.

Malicious Software

All software is potentially malicious because it tells the microprocessor to do something that it wouldn't do otherwise if it were just left alone. Antivirus and Trojan detection software that is designed to distinguish between well known malicious code and everything else is an

imperfect defense against all types of malicious code simply because there's no way for antivirus software vendors to know in advance and automate detection of every assortment of machine code instructions that can possibly cause undesired results or destroy and steal data. Malicious is in the eye of the beholder to a certain extent. The fact that there are multiple ways for machine code to wind up inside a microprocessor, such as the buffer overflow vulnerability discussed previously, that are difficult for antivirus software makers to protect against due to technical limitations of conventional programmable computers, it may be impossible to keep out all malicious code all of the time. This forces defense at a variety of levels to detect and contain the damage that malicious code unleashes when incidents occur.

Hacking Tools

Hacking tools, by themselves, are not necessarily malicious software. Many hacking tools were created by white hat hackers to enable them to prove security for a network or an individual box. What makes a hacking tool malicious is the reason it gets used and by whom against whom. An important part of information security is having access to, and routinely using, hacking tools to find out whether a black hat hacker would have any success using such tools against you. This type of white hat hacking can form the basis of preventative penetration testing but more importantly it is also useful for discovering penetrations that may already have occurred without detection. Whenever you use tools that black hats might use to see what they would see and probe your own vulnerabilities there's a chance you'll notice symptoms of intrusion or malicious attacks that were missed before. The best source of information and instruction for white hat hacking and penetration testing is the Internet. Members of the infosec community and black hats themselves provide endless instruction and sample code. You can spend months at the task and only scratch the surface. You may conclude that attempting to do everything yourself is impractical and that outsourcing penetration testing to infosec firms that specialize in this area provides a better return on your time and resource investment.

Password Crackers

Password crackers are an automated threat that stems from weak passwords or weak password protection in password databases. Some password crackers attempt to guess, or brute force, a password through repeated attempts to authenticate. Any system that allows an unlimited number of authentication failures without implementing some sort of brute force password cracking countermeasure, such as temporarily locking out the affected user account, leaves itself highly vulnerable to this type of attack. Brute forcing passwords is easiest if the passwords are weak, which means they are short, predictable, contain only words found in a dictionary, or contain personal information such as the user's birthdate or the names of their children. Other password crackers decrypt password databases or require interception of encrypted passwords as they are transmitted to password-protected services.

Consider the following scenario. A malicious attacker sits down at a Windows server box where the administrator has already logged-in. The attacker has 30 seconds to do something to compromise the box without being detected and the administrator keeps hash codes of the entire system so any malicious code that is placed on the box will give away the intrusion. What can the attacker do that will compromise the box for a later time

without leaving any trace? Attempt to access a password-protected resource such as a file share on the network and capture the network traffic that results using a network sniffer. Using a password cracker that repeatedly hashes random passwords until a match is found for the hash value sent by the box when it attempted to authenticate with the password-protected network file share the attacker can discover the administrator password and return later, either physically or through a remote login, with administrative privileges.

Port Scanners

TCP/IP networks advertise the availability of services for clients and peers through the use of port numbers, which are also used to allow a single node on the network to engage in thousands of simultaneous communications through virtual circuits identified by port number at either end. IP addresses are used for inter-node routing whereas port numbers are used for intra-node routing, that is, to allow a node that receives packets from the network to route the packets to appropriate software running locally on the node. A common black hat tactic is to scan for port numbers on a target node that the node makes available for clients and peers to use when initiating communications. Port scanners can be partially blocked if they target a range of ports on your IIS box, but they can't be completely blocked when they are designed to search only for one port, such as the HTTP port 80, because there is no way to distinguish between a port scan of port 80 that randomly finds your server's IP address and a legitimate request from an authentic client. A port scan may be the first stage of an attack or it can be a harmless part of network mapping by crawlers, some of which serve useful security and incident response purposes. However, it is a good idea to protect your entire network from port scanning and incorporate awareness of it into your intrusion detection system.

Worms and Viruses

Not all malicious code is self-replicating. In fact, most malicious code or code that is used for malicious reasons such as the hacking and information warfare tools just mentioned can't self-replicate. A program that passively self-replicates by infecting files or a storage device like a hard drive's boot sector can only infect other systems when infected files or storage devices are installed and used on other systems. These passively-replicating infectious programs are called viruses. Worms, on the other hand, actively self-replicate. They seek out other computers to infect and actively attack them, seeking a way in much the same way a malicious black hat hacker does. Viruses can usually be avoided with common sense safe computing practices because they can't penetrate a system without help from the user. Carefully antivirus scanning files before they are used and avoiding untrustworthy files to begin with are the foundation of safe computing. Worms, however, can't be prevented through safe computing alone because they exploit security holes that allow them to propagate automatically through computer networks. Defense against worms requires multi-faceted countermeasures that include elements of security policy, intrusion detection, process and network usage auditing, and coordinated incident response.

Trojan Horse Programs

A Trojan horse program is one that enters your system under false pretenses or with complete secrecy and then lurks, waiting to do something malicious until some event occurs or a period of time elapses. Any program can contain a Trojan horse, even authentic versions of retail software. The only way to know for sure that your computer isn't compromised by a Trojan is to review every line of source code yourself, or trust somebody else who has reviewed the source code, and ensure that your computer only ever runs authentic uncompromised compiled code that is known to have been compiled from the source code as it was reviewed. Clearly a daunting task, and not one that any single person can expect to undertake. This places additional burdens on software vendors to issue hash codes for their software and use digital signatures to certify authentic hash codes. At some point software developers must be trusted so developers you choose to trust must be trustworthy to begin with.

This brings up an important point. Software development tools can also be compromised by or replaced with Trojans. Think about this question: how would you know if your compiler had been replaced with a Trojan that inserted malicious machine code in the compiled object modules you produce? Or if you're a Web developer or administrator, how would you know if the file copy command you use to deploy files to the server was really a Trojan that inserts its own hidden Web page content or server side script? Perhaps you would compute the hash code of the compiler binary file before each compilation or compare hash codes on application files before and after the files are copied to the server. In that case how would you know that the program you use to verify hash codes isn't itself a Trojan? Perhaps you visually inspect a hexadecimal dump of the hash verification program before you run it. Well then how do you know the program you use to produce that hex dump isn't a Trojan, and for any of the above rhetorical questions how do you know the operating system itself hasn't been compromised with a Trojan? When operating system files or privileged programs that run as part of the system are compromised by Trojans they are referred to as a rootkit. The most threatening type of rootkit is a stealth rootkit that has the ability to hide its own existence by altering the functionality of code designed to detect the presence of the rootkit. How do you know for sure that your system isn't compromised by a stealth rootkit?

The short answer is you don't know with absolute certainty and at some point you have to close your eyes and hope for the best. Read-only operating system images that have been validated against known trustworthy binaries combined with physical protection of the box that houses the operating system image and the motherboard, memory, CPU, and wires that connect these and other pieces of the computer together is a good start. But this requires special equipment. Off-the-shelf generic PC hardware won't suffice, and Windows needs special help from software add-ons in order to run properly from a read-only boot device. For now, until improved infosec technology becomes more prevalent, the practical answer is a combination of periodic forensic analysis of your IIS boxes including hash code verification of application and OS files and detailed scrutiny of security logs. If you can't completely prevent Trojans and rootkits from threatening your programmable computers, and the fact is you just can't, then you can at least be prepared to detect them reliably through identification of unauthorized foreign code and symptoms of Trojan activity. You can also assume that your IIS box will, at some point, be compromised by a Trojan and have an incident response plan ready that will permit damage control and system integrity reassurance.

A classic Trojan will function as a backdoor to let an unauthorized attacker access the compromised computer later. Backdoors that are implemented as stealth rootkits are especially dangerous, as they have the potential to cover their tracks by erasing nearly all evidence of local malicious activity. This leaves network traffic monitoring as one of the only types of forensic evidence of penetration by a backdoor stealth rootkit, since an attacker will need to contact the infected box remotely to take advantage of the backdoor. But by that time the backdoor has already been exploited, and there's no guarantee that you'll be able to pick out the malicious network traffic produced by use of the backdoor. It will probably be encrypted and can be made to look very much like SSL-encrypted Web browsing from the perspective of a network analyzer. Periodic memory dumps with a system-level debugger can spot a backdoor stealth rootkit reliably, if you know what to look for. This means you need a baseline trusted memory dump against which to compare. And the effort required to conduct this type of forensic analysis may not be justified to guard against the type of damage that would be done if a stealth rootkit does end up on your box. Part of your security policy, disaster recovery, and risk management plan is developing a realistic assessment of the value of the data you must protect so that you can select an appropriate level of paranoia and conduct your security audits accordingly.

Web Crawlers and Search Engines

A Web crawler is a software robot that automatically visits the sites hosted by your IIS box. The robot may have benevolent intentions like helping site visitors locate Web pages by incorporating the sites' content into a search engine database but there are few other benevolent reasons for crawlers and many undesirable reasons. Crawlers crawl in order to steal content such as e-mail addresses for delivering unsolicited commercial e-mail, junk e-mail, or spam. They crawl in order to map vulnerable networks or compile lists of attack targets. They crawl to find information about specific people or companies. They can even crawl to cause DoS conditions.

Even benevolent crawlers need to be reigned-in so that they don't accidentally cause harm. It only takes one leaked private URL published to a Web site somewhere to send crawlers scrounging through private, unprotected administrative directories. The first defense is to keep all administrative directories properly password protected. But that, in and of itself, only creates another target for attackers to throw resources at if they discover the existence of the password protected private URL. Malicious crawlers may search for password protected URLs purposefully in order to find sites that have something to hide. Preventing private URLs from leaking into the public can be partially accomplished with the use of a simple text file named robots.txt that tells well-behaved crawlers what content they have permission to retrieve. The robots.txt file is placed in the root folder of each FQDN-based Web site instance, not each subdirectory-hosted site in an FQDN, and it is structured as follows:

```
User-agent: *  
Disallow: /
```

Where User-agent: can contain the known user agent string provided by a Web crawler robot in its HTTP requests. The User-agent: * shown indicates that the Disallow: rule applies to all robots. You can list Disallow: rules one line at a time to explicitly disallow certain files

and directories from being traversed by well-behaved crawlers that pay attention to the contents of robots.txt. In addition to robots.txt there is an HTML <META> tag that can influence the behavior of well-behaved robots. Any well-behaved robot that encounters the following <META> tag will avoid crawling through the content of the HTML document and won't traverse its links.

```
<META name="ROBOTS" content="NOINDEX, NOFOLLOW">
```

Crawlers are also known to crawl for intelligence and archival purposes. Information published on the Web in the past can be of significant importance in the future for civil or criminal legal proceedings and investigations, national security, or other less-than-malicious reasons. Of course, your definition of malicious may include government, law enforcement, or court review of Web site content, in which case you should turn off your Web site immediately and go do something else with your time. Automated Web crawlers are a reality, and they don't necessarily identify themselves as crawlers, especially if they border on or cross into being malicious. Automatically detecting and blocking ill-mannered robot crawlers will at least reduce the load on your IIS boxes and may protect them from some intrusions or disruptions.

Software Bugs

Software bugs are truly ubiquitous. Virtually every program ever written has bugs. Sometimes the bugs don't impact a program's operation until a variety of factors converge, such as the installation of additional software that interferes with the program for reasons that could have been avoided if the program didn't have the bug. Most bugs are dismissed by programmers and software vendors as practical limitations resulting from design decisions. A program written for Windows 95 that doesn't work properly under Windows 2000 can be explained away as a platform incompatibility, even if there's no good technical reason for the program to fail to function properly under Windows 2000. The number of reasonable-sounding excuses for programs to misbehave by design can literally exceed the number of people using certain programs, so many software vendors never get around to improving buggy code. Users are free to stop using a program that doesn't do anything useful for them, and product liability laws don't currently compel programmers to meet minimum standards of quality or refund customers' money and potentially reimburse them for damages done by their buggy software. It's difficult to imagine any law that could capture accurately the technical definition of a bug, since software bugs can be even more subjective than assessments of whether or not a program behaves maliciously. When a program's behavior, or more importantly when an entire category of programs that all behave similarly, qualifies as a security flaw is when infosec gets involved and vendors are pressured to release bug fixes. Bugs that make your life more difficult may not qualify as real bugs if they don't threaten your data security. Understanding the difference can help you communicate effectively with software vendors to get real problems taken care of promptly.

Network Security Holes

Computer networks are subject to a variety of active threats. A TCP/IP network is in some ways more vulnerable than networks that use other protocols because a TCP/IP network is designed to facilitate packet routing across long distances. The moment packets leave

the immediate physical vicinity of a node's own local area network, they cease to be trustworthy and must be authenticated, checked for tampering, and assumed to have been intercepted and recorded by a third party by the time they reach the destination. A network without routers that doesn't span long distances can be assumed to be as secure as the physical premises that protect the computers themselves. No safe assumptions can be made about the security of a TCP/IP network that routes traffic between networks. Every bit in every packet sent or received can be maliciously forged.

Only the application of cryptography to computer networking enables the information sent and received across a wide area network to become trustworthy. And even then only to the extent that it is reasonable to trust that the remote node has not been itself maliciously compromised. The use of encryption alone is insufficient to create trust, as encryption is just one part of cryptography. It takes digital signatures, certificates that certify trustworthiness and authenticity of digital signatures, hash algorithms, and other cryptographic systems combined with adequate security practice and technology to authenticate identities, prevent impersonations, provide a measure of privacy, and establish reliable trust. Further, all technological security systems must be backed up with human common sense and a diligent perpetual search for any evidence suggesting trust has been compromised.

Man in The Middle Vulnerabilities

The Man in The Middle (MITM) is a pervasive security threat that threatens all digital communications. Every digital system is potentially vulnerable to a MITM attack. Any time a third party, or malicious technology, can get in between you and the person, organization, or computer you interact with and impersonate both sides of a transaction there is said to be a MITM vulnerability. Many analog systems are also vulnerable to a MITM. The MITM can exist at any point in the process of communication whether or not computer technology is used to facilitate the communication. Your local post office is a MITM, able to open all your mail and then print, fold, stuff, lick, and stamp envelopes that appear to come from the people who send you postal mail but in fact contain forged communications. The MITM threat is one everyone lives with in the physical world, and things tend to function okay anyway. The implications for digital communications are more severe, however, because of the ease with which a MITM can get in-the-middle and near impossibility, given the limitations of current technology, of detecting their presence when they do.

DNS Hijacking and Spoofing

DNS is the single largest threat to Internet security and for now there's little that can be done about it. When a Web browser attempts to contact your IIS box using a fully qualified domain name (FQDN) it relies on a DNS server elsewhere on the network under somebody else's control. The DNS server may or may not bother to query the authoritative DNS server for the domain by which your IIS box is accessed, but regardless the end user's client browser has no choice but to assume the information it receives from the DNS is trustworthy. And the browser makes this assumption based on information it receives in an unencrypted UDP packet that the DNS server sends in response to the browser's DNS lookup query. There is no way for the browser to know if the DNS server was hijacked, and further there is no way for the DNS server to know if

the DNS server it in turn relied on was hijacked. Spoofing, where bogus DNS data is sent to clients and servers or a malicious DNS server masquerades as an authentic server to intercept and service its incoming requests, is trivial and there's almost no way to detect when it happens. Defending against insecurity in DNS means, for some applications, that IP addresses must be used instead and the DNS essentially abandoned. Consider the threat that domain-based hyperlinks represent when DNS is hijacked or spoofed. Any tampering with DNS can divert Web users to any IP address of the attacker's choice, and every domain-based hyperlink anywhere, including all search engines, printed brochures and business cards, all lists of links such as bookmarks or favorites, and every memorized URL creates new victims.

Proxy Farms and Cache

Cache is a subtle but important threat that requires constant vigilance and offers few countermeasures. Cache is everywhere in a computer network, from the CPU on the client to the CPU on the server and potentially anywhere in between. When cache malfunctions or functions in a manner that is inconsistent with your application's requirements it causes problems and can undermine security. When IIS serve client requests, a server-side cache is consulted whenever possible. Depending on the type of content being served the cache may contain a copy of the script that a particular script engine will process to produce dynamic content, so that IIS need not reread the script from disk to process each request, or the cache may contain a static copy of the output preformatted and ready to deliver to the browser client. If dynamic content is ever mistaken for static content, stale preformatted output may be delivered inappropriately to clients which denies those clients access to the authentic content. When new content is published to the server, datetime stamps on content files are compared by IIS to datetime stamps on corresponding cache entries and only newer files based on datetime stamps are refreshed in the server-side cache. This can cause unexpected behavior and break hosted Web applications in situations where system clocks are out of sync, or in different time zones, on development and production boxes so that recently deployed production files end up with datetime stamps that predate those from versions of those same files already cached by IIS.

Managing IIS cache isn't difficult, it just requires awareness of cache settings and attention to details like datetime stamps. The real cache problem, the one that leads to serious security flaws, is one you have little or no control over: proxy cache. Proxies that sit between clients and servers and mediate client access to servers are commonly designed to cache content so that additional round trips to servers can be avoided in order to preserve bandwidth and improve performance at the client. Some proxies are designed to be aggressive in their cache management policy, preferring cached data over updated data in certain circumstances even when updated data is available from the server. This results in clients receiving stale content even after IIS have been updated successfully with new content. Worst of all, it can and does result in multiple clients being sent the same Set-Cookie HTTP headers, since the proxy would have to make another round trip to the server in order to obtain a fresh cookie for the new client request and that extra round trip is expressly what the proxy is designed and configured to prevent.

In an ideal world proxies don't cache cookies and give them out to multiple clients, but that implies that in an ideal world proxies don't cache and the reality is that they do. A well-

behaved proxy will never cache cookies and will only cache large files with MIME types that aren't normally associated with dynamic content. A well-behaved proxy will also check for updated versions of cached files with every request, a process that doesn't require the proxy to retrieve the response body from the server but only the HTTP headers, resulting in time and bandwidth savings which are the purpose behind the proxy in the first place. The proxy issue is more complicated than expecting individual proxies to be well-behaved, unfortunately. Large networks commonly rely on proxy farms, collections of proxy servers that all work in unison to balance the processing load of large numbers of simultaneous users. Sometimes individual proxy servers in a farm serve responses out of cache while others do not. This can happen because of configuration mistakes or it could be by design so that some number of proxies in the farm always prefer the cache. Consider what happens, also, when more than one proxy cache exists between the end user client and the authentic server. If the proxy nearest to the client isn't aware that its own requests are routed through a proxy, the proxy may be unable to contact the authentic server directly to determine whether updated content is available. Proxy farms, proxy farm chains, cache configuration mistakes, incorrect datetime stamps, and ill-behaved caching software anywhere on the network, including the browser software on the client, create a complex web of cached content that only you can navigate. End users aren't supposed to be concerned with cache, so you have to plan ahead so that they are never adversely impacted as a result of it, even when it misbehaves. There are several programming and IIS configuration cache-busting countermeasures possible. Deploying them is a necessary part of securing IIS as a production application host, but optional if IIS will only serve static content without cookies and accurate usage tracking isn't an important design requirement.

Privacy Leaks

A processor serial number is a good example of a design feature that can be viewed as either a vulnerability, if your primary concern is privacy protection in public data networks, or a security feature, if your primary concern is enabling recovery of stolen property or deploying a private network of secure computers where a unique hardware identifier in each box is one of your design requirements. There are many ways to look at such issues, including by analyzing the difficulty with which a malicious third party might forge or intercept a particular serial number and thereby compromise the integrity of systems that might rely on it for authentication or encryption. The existence of a serial number that can be verified by an administrator as part of a routine security audit, or law enforcement when stolen property is recovered, is arguably a good thing since it gives administrators another point of assurance that can be used to confirm that authentic hardware hasn't been swapped with malicious hardware and gives property owners a way to prove ownership and recover stolen goods. For the serial number to be accessible to software running on the box, however, implies that there is a real-world practical need on the part of programmers for access to this information.

To rely on any information embedded in hardware as authentic the software that accesses and uses the information must be tamper-proof. It also must have a way to verify the authenticity of the information it receives from the hardware in addition to verifying its own integrity. The technology to accomplish this secure bridge across hardware and software has not yet been built, although some of it has been designed. Without extra data security layers, a processor serial number that is accessible to software is no better than

random numbers, encryption keys, and unique identifiers created and stored by software. Software generated numbers may even provide superior capabilities for security applications, and they're easier to change periodically which is an important part of key management. This leaves only two practical applications that are enabled by a software accessible processor serial number: backdoors to disable features or software that a vendor no longer wishes to allow a particular box to utilize, and computer forensics applications like activity tracking where confiscated hardware can be matched up with logs in which the processor serial number or a number provably generated using the serial number is present detailing actions allegedly taken by the confiscated hardware. Viewed another way, these two applications may enable better software licensing controls and better usage profiling on computer networks including the potential for network routers and servers to associate users' processor serial numbers with known identities based on information provided by ISPs, credit card issuers, or the users themselves. From most end users' perspectives neither of these new capabilities make the world a better place and are viewed as a data security threat.

Unintended publishing ahead of schedule is a common occurrence that results in a privacy leak as potentially harmful as a security breach. Safeguards that require explicit approval and authorization for new content or applications to be published help prevent premature publishing and also protect against third-party hijacking of IIS publishing points. Control of IIS publishing points should not be limited to just publishing access restrictions but should also force every publishing point to comply with a secure publishing procedure that enforces data security and privacy policy.

The Human Threat

Social engineering is the term used by infosec types and black hats to describe the vulnerability that each person in a position of trust and responsibility represents. A social engineering exploit is one that convinces a human to do something or give out information that helps an attacker mount an attack. Like authenticating with a computer system that requires a certain cryptographic protocol and will reject bogus data, humans need to be convinced that the request they receive is legitimate, and in many cases this requires only a certain manner of speech and awareness of the policies, procedures, and personnel in an organization. Small organizations are less vulnerable to social engineering than are large organizations, simply because each person knows each other and would recognize an imposter on the telephone or in person. Would a member of your small organization be able to identify as a forgery an e-mail message that appears to come from you, and appears to be asking your coworker to do something routine that you've asked them to do via e-mail before? Possibly not. That's social engineering, and it's a very real and very serious threat.

Script Kiddies

This is the era of the global teenager. There are legions of young people around the world who have Internet access and know more than you do about what the black hat hacker community is busy doing. Many of these kids possess enough programming ability to create malicious software. It's a good bet the majority of malicious code running around in the wild was created by a teenager. That is if the rumors about antivirus software makers' culpability for the code are in fact false. Hacking tools whose sole purpose is to

exploit vulnerabilities and cause harm or give control of a box to an attacker are widely distributed amongst teenage hacker groups regardless of who created them. These young people, known as script kiddies by infosec experts, represent a distinct and powerful threat and may be the first to act upon announcements of the discovery of new vulnerabilities.

The fact that script kiddies can move faster to attack your IIS box than you can move to protect it from new threats gives rise to a simple data security truth: if your box is the target of well-informed malicious attack you may be unable to defend it successfully without the help of a better-informed managed security monitoring service.

Script kiddies aren't so much the threat as they are hands that set threats in motion. The threat is simply that programmable computers are programmable. If they can talk to each other, they may be able to reprogram each other. If a user can access a programmable computer, the user may be able to reprogram it. Script kiddies apply the programmable computers they control for malicious purposes, and escalate and renew threats by keeping malicious code running on as many computers as possible, but if they didn't exist the potential threat would be the same even if the effective threat would be lower on a given day. Self-replicating worms and viruses are better at mounting distributed coordinated attacks than are script kiddies, but when the two are combined they feed off each other and grow out of control. To defend against script kiddies you have to defend not only against all known threats by protecting all known vulnerabilities but you also have to defend against all unknown threats by protecting all potential vulnerabilities. One of the best ways to accomplish this is to configure an explicit restrictive security policy on your IIS box that will allow it to execute only software that it is authorized to execute and process only requests that it is authorized to process while rejecting everything else.

Rogue Employees and Sabotage

Data can be destroyed. Your own employees can turn against you. People who are trustworthy now may not be later, but your organization may not detect the change before it's too late. One of the only protections possible against threats of this level is transaction processing. Whenever possible, implement policies and procedures for updating IIS and its content that identify who in your organization took certain actions. Even if you can't back out the actions of a rogue employee with a single mouse click, if you have a transaction log showing the work they did you can at least mount some sort of security response. Manage your development, deployment, and system updates with awareness of the worst-case scenario, and plan in advance for this type of incident response. The very existence of safeguards against sabotage and rogue employee tampering may be enough to deter it in the first place. An angry employee, or ex-employee, who knows they will be caught if they take a malicious action will possibly think twice about attempting to cause any harm. Always plan ahead for password changes when employees leave the company under any circumstances. Ideally you would change all passwords every morning. If this isn't practical, redesign your IIS deployments and security policies so that it can be.

Eavesdropping

Somebody else is always listening to everything your IIS box says. Assume that every bit it sends on the network and every screen full of information it displays is leaking out to a third party who is recording it all. What security risks would this create? Eavesdropping still occurs even when every bit of information is strongly encrypted. Whoever the eavesdropper is may some day discover the key and be able to decrypt all of the encrypted communications they intercepted over the years. When you build secure systems around IIS, you must assume that encryption keys will be compromised and design regular key changes into your applications. Destroying keys when they will no longer be used is important, but you can't do anything to stop a sufficiently-determined attacker from breaking encryption through cryptanalysis given enough time to work on the problem. The only real security you have against eavesdropping, even when encryption is used, is regular key changes. This prevents trivial eavesdropping that any script kiddie can accomplish and it also locks out even very well-equipped determined attackers. Some of your private communications may be decrypted, but hopefully not enough to do significant harm. And, the more time it takes an attacker to complete cryptanalysis the less relevant the information could be, anyway, if any of it is time-sensitive. Keeping a malicious third-party from accessing data for three days that becomes useless or gets released to the public after two days is the equivalent of preventing an eavesdropping attack completely.

Unnatural Disasters

There are a variety of unnatural disasters that could impact your IIS box. If it isn't completely destroyed to a degree that prevents any forensic analysis and data recovery from being possible then the unnatural disaster may represent a security threat. Political upheaval, large-scale violent crime, vicious lawsuits, corporate espionage, and budget cuts. Some thought should be given to these worst-case scenarios because they may be as likely to happen as any natural disaster. In times of crisis, you will be the one responsible to ensure security and integrity of IIS.

Welcome to the infosec profession.

Chapter 2: Microsoft Internet Information Services

This chapter answers the question “what are Microsoft Internet Information Services?” There is no single answer to this question but there are technically accurate ones. To properly secure IIS you must understand what these services really are behind the high-level summary description most often given of IIS. If you think of IIS as the “Microsoft Web services for Windows NT/2000/.NET Server” you’ll be able to communicate clearly with most of your peers but you’ll be perpetuating a myth/understanding (that’s a misunderstanding derived from a myth) that undermines your ability to build, deploy, and maintain secure TCP/IP network information systems built around Microsoft Windows Server operating systems.

Instead of just giving descriptions of IIS, this chapter shows you their guts and lets you draw your own conclusions or create your own mythology to describe them. What you need to remember as you read this chapter and indeed the rest of this book is that from a network traffic perspective there is no such thing as software. The right sequence of bits hitting your box can take control of your hardware by exploiting security flaws or spoofing, and software plays a central role in that, but your ability to secure data and protect networks is increased when you let go of software myopia. Insecurity is a default feature of all network software and hardware. Good network security requires every point of vulnerability to assume that malicious programs or people are trying to do bad things with bits on the network. If you don’t want to or can’t properly manage security on a box, don’t allow it to receive bits from the network that it didn’t request while acting as a client or a peer.

All of the information services provided by IIS share three common features: TCP/IP, dynamic content, and programmability. If you choose to run any part of IIS, you have an obligation to properly manage and configure the TCP/IP settings, dynamic content, and programmability interfaces the services expose. By following recommended IIS security best practices detailed in this book you will achieve your desired results while controlling security impacts and unintended consequences. The key to accomplishing these objectives quickly and easily is to start with low-level knowledge of the design decisions made by Microsoft programmers when they coded the IIS architecture.

Architecture Design

Data arrives from a sending device on a TCP/IP network in packets addressed with a local port number and a remote port number. When the local port number matches one that your Windows Server operating system has been configured to associate with Internet Information Services there is a possibility that the data represents an inbound communication to which it is appropriate or necessary to send a response. IIS provides machine code instructions and thereby controls CPU cycles, memory, and any I/O hardware required to decode the inbound communication and transmit an encoded response by way of TCP/IP packets addressed with the inverse local and remote port number pair. The original sending device becomes the receiving device and is able to distinguish the response packets from other TCP/IP network traffic by matching port number and IP address with its own in-memory table of port numbers and associated

applications. This is the starting point of any TCP/IP server application. IIS grew from this starting point to be the TCP/IP application services development platform they are today because of Microsoft's decision to create more than just Web services but rather to create a platform for building interactive TCP/IP network services around a single managed architecture.

Microsoft decided to enable other programmers to design and configure systems that encode and decode application data on a TCP/IP network using any conceivable digital algorithm or standard within a single server programming framework. The decision was made to tackle a larger problem and build richer features than were minimally necessary to implement TCP/IP network application protocols like HTTP. Thanks to this decision, the IIS codebase evolved into what it is today and its services are now enabling sophisticated network programming around XML and .NET that might have been unimaginable only a few years ago. The expertise gained by early adopters who devoted the time necessary to master secure and reliable deployment and maintenance of network applications on the IIS development platform, combined with thousands of third-party applications and code samples, make the IIS platform appealing for the same reasons that the Windows platform itself is appealing.

What IIS are by default on the first boot of a fresh installation of a Windows Server operating system just provides a convenient starting point that represents the core features necessary for TCP/IP network server application development. As you can see in Figure 2-1, IIS 6 has completely redesigned the core functionality of IIS, giving them new versatility and performance for high-volume request processing through the help of worker processes that make Local Procedure Calls into protected (Kernel mode) operating system code to send and receive data. All third-party code has been forced out of the process space created by IIS and into these worker processes making it impossible for insecure or malicious applications to impact the stability of the IIS platform.

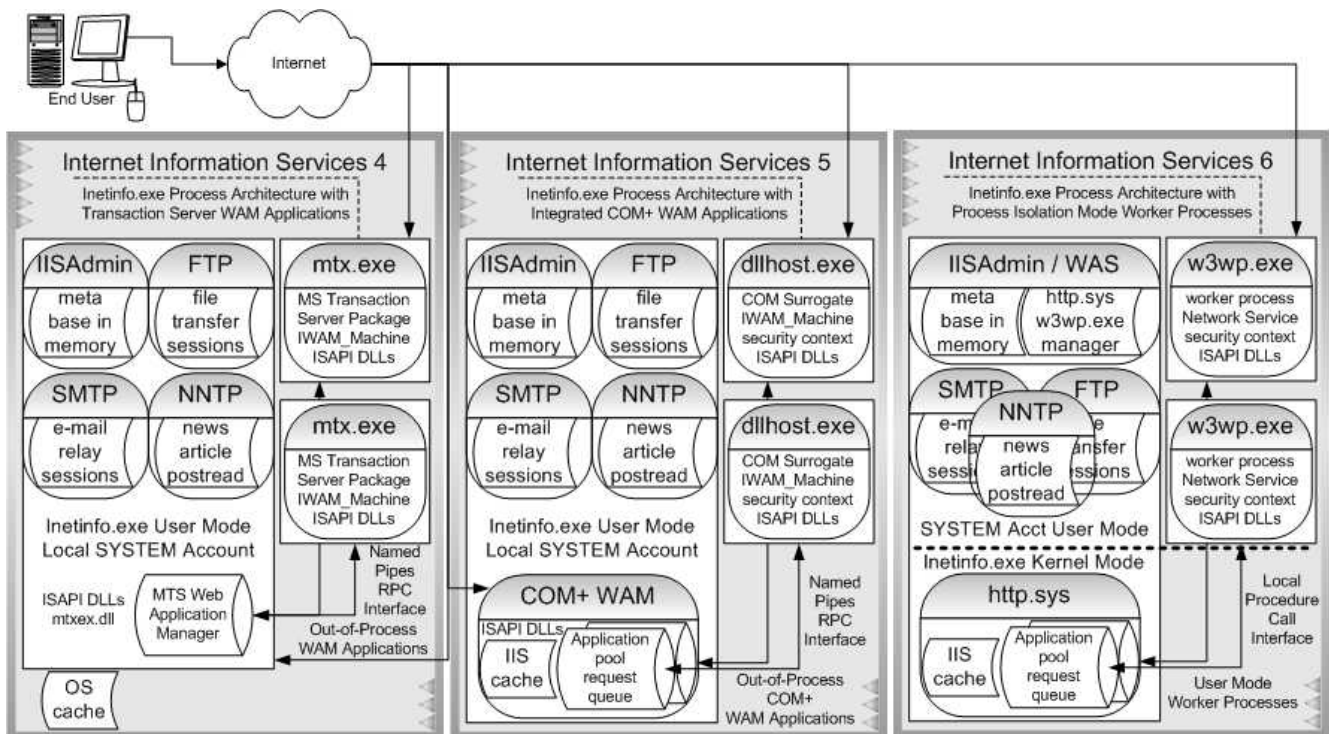


Figure 2-1: The Internet Information Services Architecture

IIS 6 allows no third-party code inside inetinfo.exe, locating it instead inside worker processes and creating application pool request queue and cache management at a kernel mode device driver level. This accomplishes in a default configuration security and performance not possible in IIS 4 and 5 out of process applications. As you can see in Figure 2-1, http.sys (whose code name was “Duct Tape” in beta) is the code module that implements the Kernel mode layer for IIS 6.

The IIS Platform

Internet Information Services provides for Windows Server operating systems a modular TCP/IP network server Application Programming Interface and Software Development Kit. Using standardized application layer protocols defined by W3C and Internet Engineering Task Force working groups as a foundation, IIS delivers manageable services platform code for hosting application layer TCP/IP network server software. The features and extensibility that make IIS challenging to secure exist in order to provide this platform for developers and represent a key strategic advantage for IIS versus other development platforms. Microsoft chose to build into Windows Servers this platform for TCP/IP development in order to complement its client-side TCP/IP development platform commonly referred to as Internet Explorer.

It doesn't help computer security professionals to think of either IIS or IE as individual software programs because that thinking, aside from being technically inaccurate, tends to lead to security blind spots where the platforms have silent impact that goes unmonitored and unsecured. It also doesn't help computer security professionals to be given responsibilities to protect and defend computer systems without security documentation, access to the source code, or a comprehensive technical analysis of the software they must trust implicitly. With IIS 6 Microsoft acknowledges these shortcomings

of previous IIS releases and ships the IIS platform as an optional operating system feature, configured by default in its most secure mode with minimal request processing abilities. Administrators choose which of the IIS modules to deploy and have more access to security architecture documentation and details of past security vulnerabilities.

inetinfo.exe

The core IIS architecture revolves around several binary modules with inetinfo.exe at the root. Under IIS 4, 5, and 6 the inetinfo.exe module executes by default under the System security context, the most privileged built-in account. This means that any buffer overflow vulnerability that can be exploited inside inetinfo.exe gains control of a process with a System security token that is afforded unlimited rights. A buffer overflow attack is simple to engineer against software that has not been hardened to prevent buffer overflows because this type of vulnerability takes advantage of the way that memory is pre-allocated for variables at compile-time as part of the stack. Any code that reads values into variables on the stack, as nearly all code does, is potentially vulnerable. Programmers have to take special precautions and use diligence and care by enforcing assumptions about the length of values loaded into stack variables in order to protect the stack from overflow attacks. Memory declared dynamically from the heap must be protected in the same way, though exploiting a heap overflow vulnerability is more difficult for the attacker. The inetinfo.exe process under IIS 4 and 5 was designed to allow the loading of third party code, and some of Microsoft's own code was vulnerable to buffer overflow attacks, and as a result the platform was not secure enough to allow for certain usage scenarios like hosting of third party code. The inetinfo.exe module under IIS 6 has been hardened meticulously and it no longer allows third party code to enter its process space, enabling IIS 6 to be used as a trustworthy computing environment for hosting of third party code.

Each module in the IIS platform exposes a Service Control Manager (SCM) interface that complies with the API used by Windows Server operating systems to enable centralized management of service process startup and shutdown through the Services Control Panel. The inetinfo.exe process itself is controlled under the name IIS Admin Service while the HTTP request processing service module of IIS is controlled under the name World Wide Web Publishing Service. Other IIS modules can be started and stopped under their own service names independent of inetinfo.exe.

Metabase

Configuration settings for IIS are stored in a file called the metabase. Under IIS 4 and 5 the default metabase file is named MetaBase.bin located in System32\inetsrv and it contains a hierarchical binary representation of all IIS settings similar in structure and function to the Windows Registry. Certain configuration settings for IIS are stored in the Windows Registry, such as the path to the metabase file and performance properties, while the metabase stores settings that determine the manner in which content is served and IIS modules accessed by clients at the file, folder, and application levels. The binary metabase can't be edited directly and Microsoft does not publish the binary file format of the metabase file. To allow direct modifications to the metabase under IIS 6 the default metabase file is an XML document named metabase.xml located in System32\inetsrv. The XML schema implemented in metabase.xml is not only published, it is a living

document that you can customize and extend named mbschema.xml also located in System32\inetsrv.

The metabase is read from disk when IIS starts up and is held in memory by metadata.dll, the IIS MetaBase DLL. Changes can be made to the metabase while in memory through the use of the Active Directory Services Interface (ADSI) which includes an IIS metabase directory access provider. Metadata.dll handles writing in-memory metabase changes periodically to the binary metabase file. Changes can also be made to the metabase file whenever the IIS Admin Service is running through the administrative interfaces it supports. The ability to reconfigure settings for any IIS module while the network service implemented by the module is stopped is an important security feature made possible by the IIS Admin Service. Under IIS 6 with its XML formatted metabase the IIS Admin Service can be configured to detect changes made to the XML metabase and automatically merge those changes with the current in-memory version of the metabase.

IIS Admin Service

After inetinfo.exe executes but before any network ports are opened for servicing incoming requests an administrative layer is activated known as the IIS Admin Service. The inetinfo.exe process implements the IIS Admin Service and acts as the host for each IIS-compatible network service module.

Internet Server Application Programming Interface (ISAPI)

Inetinfo.exe implements the core architecture for hosting Internet services application code. Each module that implements application specific logic or a runtime environment for interpreting application script is hosted within the IIS architecture as a DLL conforming to an Internet Server Application Programming Interface, ISAPI. ISAPI DLLs are designed to provide two different classes of service: Extension and Filter. An Extension ISAPI module is mapped to a list of file types and gets executed as a content handler for producing responses to requests for items of the specified type. Filters transparently layer application logic into all request processing for the Web site in which they are configured. Microsoft provides an ISAPI Extension DLL that implements a built-in server-side scripting environment for hosting application logic, Active Server Pages (ASP.DLL), that by default handles requests for .asp files.

WAM Director

The Web Application Manager (WAM) was introduced with IIS 4 when Microsoft Transaction Server (MTS) integration provided the first out of process application hosting option under IIS. MTS packages invoked within mt.exe, the host process for an MTS out of process module, by the MTS Executive (mtxex.dll) give inetinfo.exe a mechanism for hosting ISAPI DLLs and native MTS objects out of process while relying on interprocess marshaling through the Named Pipes Remote Procedure Call (RPC) interface to maintain communication with the application code. The WAM includes a Director that functions as an application message queue and router that manages interprocess marshaling or in-process dispatching to the application code that handles request processing. The other key facility provided by the WAM Director is remoting the server context with which the application code calls back into the WAM to prepare and deliver responses and obtain information about the request including HTTP headers and authentication properties.

The WAM Director sets up security account context impersonation for and spawns the threads that handle request processing both out of process and in-process. In so doing it relies on the impersonation account settings configured for IIS at the Web server root and optionally overridden by application-specific impersonation settings for each Web site and subdirectory. For out of process applications the WAM Director requests that the hosting process, an instance of mtx.exe under IIS 4, dllhost.exe under IIS 5, or w3wp.exe under IIS 6, use one of its threads to impersonate the specified security context using the token it provides. If the request is authenticated and impersonation of the authenticated identity is possible then a different token is used by the thread in order to place the thread in a security context that is appropriate for the authenticated user identity. The metabase stores the default impersonation identities for unauthenticated as well as default-, anonymous-impersonated request processing by threads. The Web server root settings are found in the following metabase locations:

```
/LM/W3SVC/WAMUserName  
/LM/W3SVC/WAMUserPass  
/LM/W3SVC/AnonymousUserName  
/LM/W3SVC/AnonymousUserPass
```

WAMUserName and WAMUserPass can be set only by editing the metabase, while AnonymousUserName and AnonymousUserPass can be set either by editing the metabase or through the Microsoft Management Console (MMC) as shown in Figure 2-2. WAMUserName and WAMUserPass are only applicable at the root W3SVC level, indicating to the WAM Director what user ID and password to use for obtaining the user account security token under which to launch out of process application host process instances.

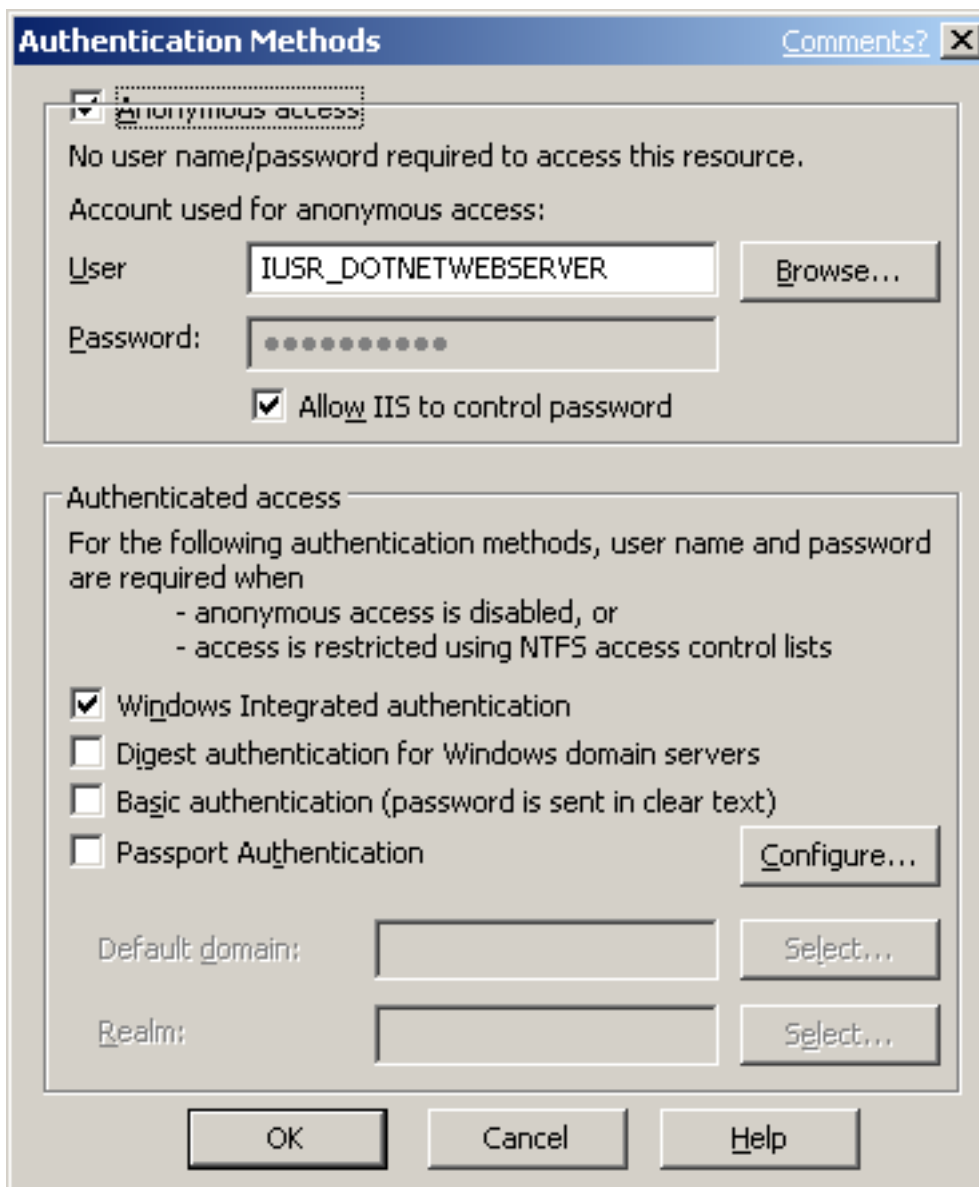


Figure 2-2: Configure Anonymous user impersonation account with MMC

Anonymous requests serviced by in-process applications have a process token that corresponds to the System account even when the per-thread security context is overridden by Anonymous user impersonation or authenticated user impersonation. The default user ID for Anonymous user impersonation in all versions of IIS is IUSR_MachineName where MachineName is the computer name assigned to the server box on which IIS are installed. Out of process request processing has a process token set by WAM Director based on the WAMUserName and WAMUserPass. The default user ID for out of process request processing in IIS 4 and IIS 5 is IWAM_MachineName where MachineName is the computer name assigned to the server box on which IIS are installed. IIS 6 provides an alternative to out of process request processing that involves application pools and enables a different security context to be configured in each application pool. Per-thread impersonation is still performed by all versions of IIS to set the effective security context of each thread in out of process application host processes.

Microsoft Transaction Server (mtx.exe)

Microsoft Transaction Server enables IIS 4 out of process applications and gives both in-process and out of process Active Server Pages (ASP.DLL) hosted application script the ability to participate in distributed transactions managed by the MTS Distributed Transaction Coordinator (DTC).

COM+ Surrogate Out of Process Host (dllhost.exe)

IIS 5 uses COM+ services for accessing out of process applications. Each out of process application is automatically configured by IIS Admin as a new COM+ service hosted by an instance of dllhost.exe that is configured to load the ISAPI DLLs required to process requests and implement Web application functionality.

IIS 6 Isolation Mode Worker Process (w3wp.exe)

Under IIS 6, the WAM Director is incorporated into the base inetinfo.exe IIS Admin Service as part of the Web Administration Service (WAS). Enhancements to the IIS architecture that include "Duct Tape", the https.sys kernel mode device driver for optimized dispatching of inbound and outbound data to and from out of process application pools, replace much of the functionality implemented by WAM in previous IIS versions. The result is superior performance and reliability and a more trustworthy foundation for the deployment of network application services. The user account under which worker processes execute in each application pool can be configured through the MMC snap-in for IIS 6 as shown in Figure 2-3.

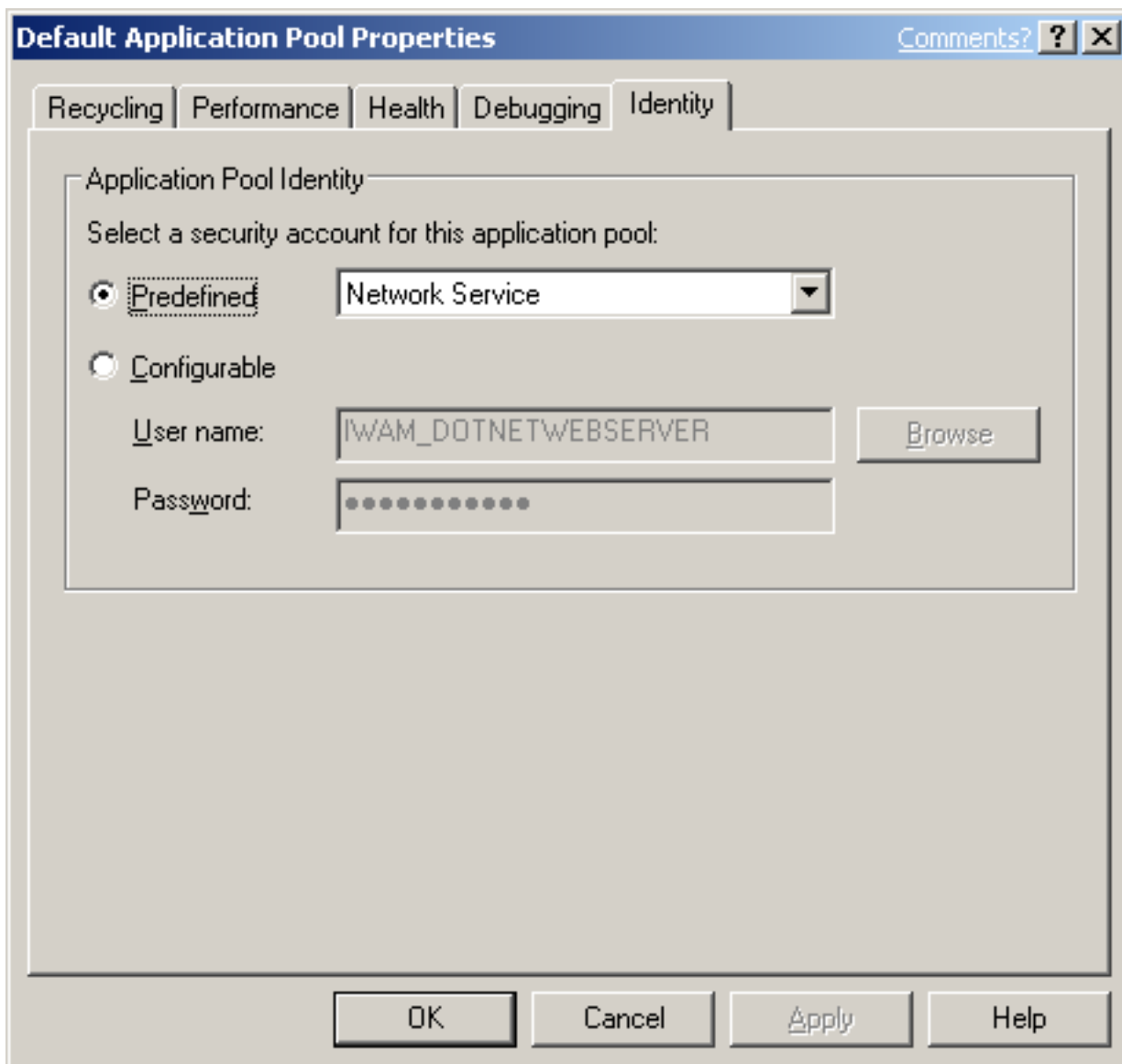


Figure 2-3: Configure account to use for each IIS 6 application pool with MMC

A Brief History of inetinfo.exe

Much turmoil has ensued in the world at large since the first build of inetinfo.exe that has directly and indirectly changed IIS forever. For example, a buffer overflow vulnerability in IIS Index Server was exploited by the Code Red and Code Red II worms, causing untold losses for companies and individuals impacted by these malicious programs. One of the interesting facts about these worm outbreaks is that the buffer overflow vulnerability they exploited was discovered and patched well in advance of the incidents that caused large-scale widespread infections and network service outages. The delay deploying the patch for this vulnerability on the millions of potentially vulnerable systems was caused in part by lack of appreciation for the severity of the risk the vulnerability represented. A continuous flow of security alerts are disseminated by system vendors including Microsoft and at the time there wasn't much in the way of a "drop everything you're doing and race to patch your servers" urgent communication channel open with security administrators inside organizations that run Windows servers. After Code Red, new emphasis was

placed on security policy throughout the computer industry and new tools for applying security-related patches were developed and IIS version 6 are the result.

Each version of IIS was shaped by the experiences and application needs of users and developers inside and outside Microsoft who used and deployed production servers using previous IIS releases. Table 2-1 lists each version of IIS and where the version came from originally. You might interpret this table something like the following:

Version 1 was released in response to the official change of policy of Internet infrastructure providers and funding suppliers (including the U.S. Government) to allow commerce to be conducted on the Internet.

Version 2 was the best effort improvements possible in time for the final release of Windows NT 4.0.

Version 3, like version 3 of other Microsoft products, represented the milestone whereafter Microsoft's platform went one step beyond the nearest competitors.

Version 4 marked the widespread recognition of the superior programmability of IIS compared to other Web services platforms and resolved stability problems.

Version 5 integrated COM+, the security and transactional enhancement to COM.

Version 6 was built using solid engineering and computer security principles ("Duct Tape") as an integrated part of the Microsoft .NET Framework to guarantee that the IIS platform can securely support every conceivable deployment scenario including the ones that ISPs and other Web hosting providers must contend with: malicious programmers in possession of passwords that control access to authorized publishing points.

Table 2-1: IIS Version History

Common Name	Description of File Origin
IIS 1.0	Optional install with Windows NT 3.51 Service Pack 3
IIS 2.0	Default install for Windows NT 4.0
IIS 3.0	Automatic upgrade in Windows NT Server 4.0 Service Pack 3
IIS 4.0	Windows NT Server 4.0 with Service Pack 3 plus optional NT Option Pack
IIS 5.0	Default install in Windows 2000 Server; optional in 2000 Professional
IIS 5.1	Optional install for Windows XP Professional
IIS 6.0	Windows .NET Server Family default or optional installation

IIS and the Windows .NET Server Family

Microsoft Internet Information Services version 6.0 are installed by default only with Windows .NET Web Server and Small Business Server. IIS 6.0 are an optional install for Windows .NET Standard Server, Windows .NET Enterprise Server, or Windows .NET Datacenter Server.

IIS send a common name version number by default in an HTTP response header named Server. For instance, IIS 5 return "Server: Microsoft-IIS/5.0" when producing HTTP responses. Automated hacking tools as well as non-malicious network clients such as the Web server survey conducted monthly by netcraft.com rely on the response headers to determine that a server runs IIS and their version number. Disabling the Web server version identifier HTTP header can be enough by itself to reduce the load on your server when a worm outbreak occurs on the Internet if the worm is designed to attack only

servers that run IIS. The default IIS version HTTP response header can be overridden by setting the following registry key and installing the SetHeader ISAPI filter that Microsoft provided with Knowledge Base article Q294735 for the entire server or for individual Web sites to override default:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC\Parameters\Server
Type

The HTTP 1.1 specification calls for the Server: header that is returned by standards-compliant HTTP 1.1 servers to be configurable. IIS do not provide a built-in feature to override default Server: header; instead an ISAPI Filter enables this:

<http://support.microsoft.com/directory/article.asp?ID=KB;EN-US;Q294735>

IIS Version 4.0

Windows NT Server 4.0 with Service Pack 3 plus optional NT Option Pack. IIS version 4.0 was the last release of IIS that was referred to in the singular. After IIS 4.0 the core value of the network service development platform, its programmability and capability to host multiple applications and go beyond the capabilities of a simple HTTP server, became the central theme of IIS. All versions after IIS 4.0 are referred to in the plural. The name didn't change; the documentation for versions 1.0 through 4.0 just had unfortunate typographical errors that made IIS appear to be a single server rather than a platform for network services.

IIS Version 5.0

IIS 5 introduced the out of process application pool and compressed files in the IIS Temporary Compressed Files folder. Optional automatic recovery was made possible when the inetinfo.exe process ends unexpectedly. A Service Control Manager (SCM) was added to the administrative interface for IIS where recovery options are specified. Options for automatically restarting inetinfo.exe, running an executable or script file, or rebooting the computer are configured using the SCM. Improved event logging gives administrators more information about performance problems. IIS 5 also provides a command-line utility, IISRESET.EXE, that simplifies creating scripted restarts of the IIS Admin Service, its hosted network application services, and the various other services that depend upon inetinfo.exe.

IIS Version 6.0

Process isolation mode and kernel mode request queueing using http.sys "Duct Tape" fundamentally change the IIS security architecture in version 6.0 and encourage adoption of the .NET Framework as a more secure programming toolkit than either ISAPI or Active Server Pages provided for previous versions. The WAM Director is superceded by this architectural reorganization and an enhanced Web Administration Service that oversees and governs http.sys and its worker processes.

Kernel Mode Request Queueing with http.sys Duct Tape

The http.sys module loaded into inetinfo.exe implements a kernel mode request queue and event-driven asynchronous communications channel into each out of process application pool where threads are available to service requests on demand and thread health is continually monitored to keep IIS responsive under heavy load. Third-party code is never loaded into the inetinfo.exe process so that nothing can interfere with http.sys in its management of request queueing and communications with worker process application pools. During development http.sys had the code name "Duct Tape" in reference to its chief technical architect's propensity to talk incessantly about http.sys even during meetings about other topics. Likewise, the http.sys module keeps IIS talking on the network no matter what else is happening around it, so the code name just seems to fit.

Multiple Worker Processes

IIS 6 include a fundamental security improvement to the process model architecture for hosting of third-party application code by the services. All Microsoft code that implements operating system or IIS functionality is now isolated from any code written by third-party developers in order to prevent such code from accessing process memory and binary modules loaded by inetinfo.exe. There is no longer any such thing as in-process application code. Instead, all application code is loaded within a security context that belongs to the Network Service built-in account with few privileges allowed to the account for access to system resources and files. All third-party code is hosted with restricted privileges in worker processes.

Process Isolation Mode

When previous version of IIS were deployed in large server farms by ISPs certain critical limitations became of central concern to them for future enhancements to IIS. Among the most important of these concerns was the need to completely isolate Web sites hosted on a single server box so that even customers with malicious intent or profit-motive reasons to gain read access to the files and folders of other customers' Web sites hosted on the same box or inside the same server farm could easily be denied that ability as a natural part of the IIS architecture.

Securing IIS 4 and 5 to this degree is complicated and goes beyond simply configuring security for IIS. For starters, every COM object that is registered on the server must be reviewed and any that might be instantiated within Active Server Pages scripts must be removed or secured with restrictive NTFS and registry permissions preventing access to its COM interface. The use of parent paths (.././folder/file) must be disabled, and several ISAPI filters must be removed because they can be misappropriated for malicious purposes to attack other sites hosted on the same server box. And every Web site must have a different anonymous impersonation account identity configured. The list of lockdown procedures goes on to encompass everything discussed in this book and more, and after they've all been implemented numerous denial of service attacks are still possible when a malicious publisher sends code to the server for the sole purpose of consuming its resources and denying access to visitors of the other sites hosted on the same server. IIS 6 solves the problem for ISPs who want to give customers full-featured IIS hosting including the ability to receive code from customers for publication on the server without worrying that customers' third-party code will maliciously attack each other or interfere with server operations. Duct Tape, worker processes, and a module known as

the Web Administration Service manager keep IIS 6 healthy and secure even under attack.

Web Administration Service

The IIS Admin Service for IIS 6 is responsible for both configuration settings through the metabase and worker process management. The IIS Admin Service encapsulates a Web Administration Service (WAS) that loads http.sys, configures application pools, and launches and monitors worker processes, recycling them when necessary. Each application hosted by IIS is associated by WAS with a particular application pool. Applications added after http.sys is initially configured by WAS at IIS startup time are added to an http.sys URL namespace routing table dynamically to enable WAS to start new worker processes as needed to service requests for the newly added applications. As application pools are added that provide additional processing capacity for servicing requests, WAS informs http.sys of the new capacity so that it can be incorporated dynamically into the application routing table.

Third-party code is never loaded into inetinfo.exe in order to protect WAS from corruption that would disrupt IIS health monitoring and automatic failure recovery service for application pools. IIS 6 isolate third-party code securely within w3wp.exe processes. This keeps critical IIS functionality including configuration management and request queue management safe from both third-party code crashes and malicious attacks by third-party in-process modules. WAS also watches for disruptions in worker processes that may be caused by malicious code and when a worker process drops its Local Procedure Call communication channel link with WAS the process is promptly recycled by WAS.

Integration with Windows Server OS

Windows Server operating systems are built according to an architectural model called microkernel. Intel-compatible microprocessors implement process privilege and memory protection levels numbered from 0 to 3. The levels are referred to as rings. With code that executes on ring 0 being the most privileged in memory that is the most protected and code that executes on ring 3 being the least privileged in memory that is least protected. The microprocessor's current privilege level (CPL) is indicated by the low order two bits of the CS register. Windows mirrors microprocessor protection levels with operating system protection layers. The most privileged or protected layer contains code that executes on ring 0 in the microprocessor and this layer is referred to as the executive or kernel mode layer. Code that executes on ring 3 in the least protected memory space is said to reside in the user mode protection layer. OS components that execute in kernel mode are protected from direct access by components in the user mode layer by a microprocessor protection interface called a gate. A general protection exception is raised by the microprocessor when code executing in user mode attempts to access code in kernel mode on a lower ring without using a protection gate. Most OS code, and almost all third-party code with the exception of certain device drivers and debugging utilities, runs in the user mode layer on ring 3 of the microprocessor. To actively manage security for IIS you need to have a solid familiarity with its organization into binary code modules and have experience working with developer tools that would be any attacker's first line of offense because those same tools are your first line of defense; they are used not to mount attacks but to gather intelligence.

Win32 and inetinfo.exe

By default, 32-bit inetinfo.exe is installed in System32\inetsrv. Its preferred memory address for its binary image when loaded into RAM by the operating system is 0x01000000 – 0x01006000. Inetinfo.exe is dependent upon DLLs that implement portions of the Win32 API as listed in Table 2-2. The DLLs listed in bold are part of the IIS Admin Service and are loaded into memory before network service modules are loaded and before TCP/IP ports are set to listen for incoming connections.

Table 2-2: Win32 DLLs in inetinfo.exe v5.0.2195.2966

DLL Filename	Preferred Memory Range	Description
System32\NTDLL.DLL	0x77F80000 – 0x77FFB000	Server OS “NT Layer”
System32\KERNEL32.DLL	0x77E80000 – 0x77F35000	Windows Base API
System32\ADVAPI32.DLL	0x77DB0000 – 0x77E0C000	Advanced Win32 Base API
System32\USER32.DLL	0x77E10000 – 0x77E74000	User API
System32\GDI32.DLL	0x77F40000 – 0x77F7C000	Graphics Device Interface
System32\OLE32.DLL	0x77A50000 – 0x77B46000	OLE for Windows
System32\ws2_32.dll	0x75030000 – 0x75043000	Windows Sockets 2
System32\ws2help.dll	0x75020000 – 0x75028000	Windows Sockets 2 Helper
System32\SHELL32.DLL	0x782F0000 – 0x78532000	Windows Shell Common
System32\comctl32.dll	0x71780000 – 0x7180A000	Common Controls Library
System32\OLEAUT32.DLL	0x779B0000 – 0x77A4B000	OLE Automation API
System32\SCHANNEL.DLL	0x78160000 – 0x78186000	TLS/SSL Security Provider
System32\secur32.dll	0x77BE0000 – 0x77BEF000	Security Support Provider Interface
System32\CRYPT32.DLL	0x77440000 – 0x774B5000	Crypto API32
System32\wssock32.dll	0x75050000 – 0x75058000	Windows Socket 32-Bit
system32\RSABASE.DLL	0x7CA0000 – 0x7CA22000	Microsoft Base Cryptographic Provider
System32\security.dll	0x75500000 – 0x75504000	Security Support Provider Interface
System32\NETAPI32.DLL	0x75170000 – 0x751BF000	Network Management API
System32\version.dll	0x77820000 – 0x77827000	Version Checking and File Installation Libraries
System32\mswsock.dll	0x74FF0000 – 0x75002000	Microsoft Windows Sockets Extensions
System32\WINTRUST.DLL	0x76930000 – 0x7695B000	Microsoft Trust Verification APIs
System32\RNDR20.DLL	0x785C0000 – 0x785CC000	Windows Sockets 2 NameSpace
System32\odbc32.dll	0x1F7B0000 – 0x1F7E1000	Open Database Connectivity Driver Manager
System32\COMDLG32.DLL	0x76B30000 – 0x76B6E000	Win32 Common Dialogs
System32\odbcint.dll	0x1F850000 – 0x1F866000	ODBC Resources
System32\wshnetbs.dll	0x754B0000 – 0x754B5000	Netbios Windows Sockets Helper

System32\odbccp32.dll 0x1F800000 – 0x1F818000 Microsoft Data Access ODBC
Installer

The 64-bit version of IIS rely similarly on Win64, the 64-bit version of the Win32 API. The Win64 API modules loaded into the 64-bit inetinfo.exe can be discovered through an audit of the process using a debugger. The module list is nearly identical, as the difference between 32-bit and 64-bit Windows APIs are minor. Auditing your IIS platform at the binary module level is an important security policy measure that should be conducted regularly on all production servers.

Auditing inetinfo.exe

You can use the Microsoft Windows Debugger WinDbg version that is provided with the Windows 2000 Service Pack 2 debug symbols to safely audit the inetinfo.exe process. Unlike previous versions of WinDbg, the new version supports noninvasive debugging of processes including system processes like inetinfo.exe. With noninvasive debugging, WinDbg can attach at run-time to an existing process to examine the process as it executes. When WinDbg does a noninvasive attach, it temporarily suspends the threads of the target process. As long as you detach from the process by choosing the Stop Debugging option from the Debug menu, request processing in inetinfo.exe continues when its threads resume just as though the debugger had never suspended its threads in the first place.

WinDbg provided with Windows 2000 Service Pack 2 debug symbols, and WinDbg versions released subsequent to it as part of DDK or SDK updates, can also be used under Windows NT 4 to conduct noninvasive debugging sessions.

Noninvasive debugging is an important tool for security administration because it gives you the ability to audit the binary code that IIS load to carry out client request servicing. Figure 2-4 shows WinDbg in the process of attaching noninvasively to inetinfo.exe under Windows 2000. The hotkey F6 or Attach to a Process from the File menu brings up the Attach to Process window.

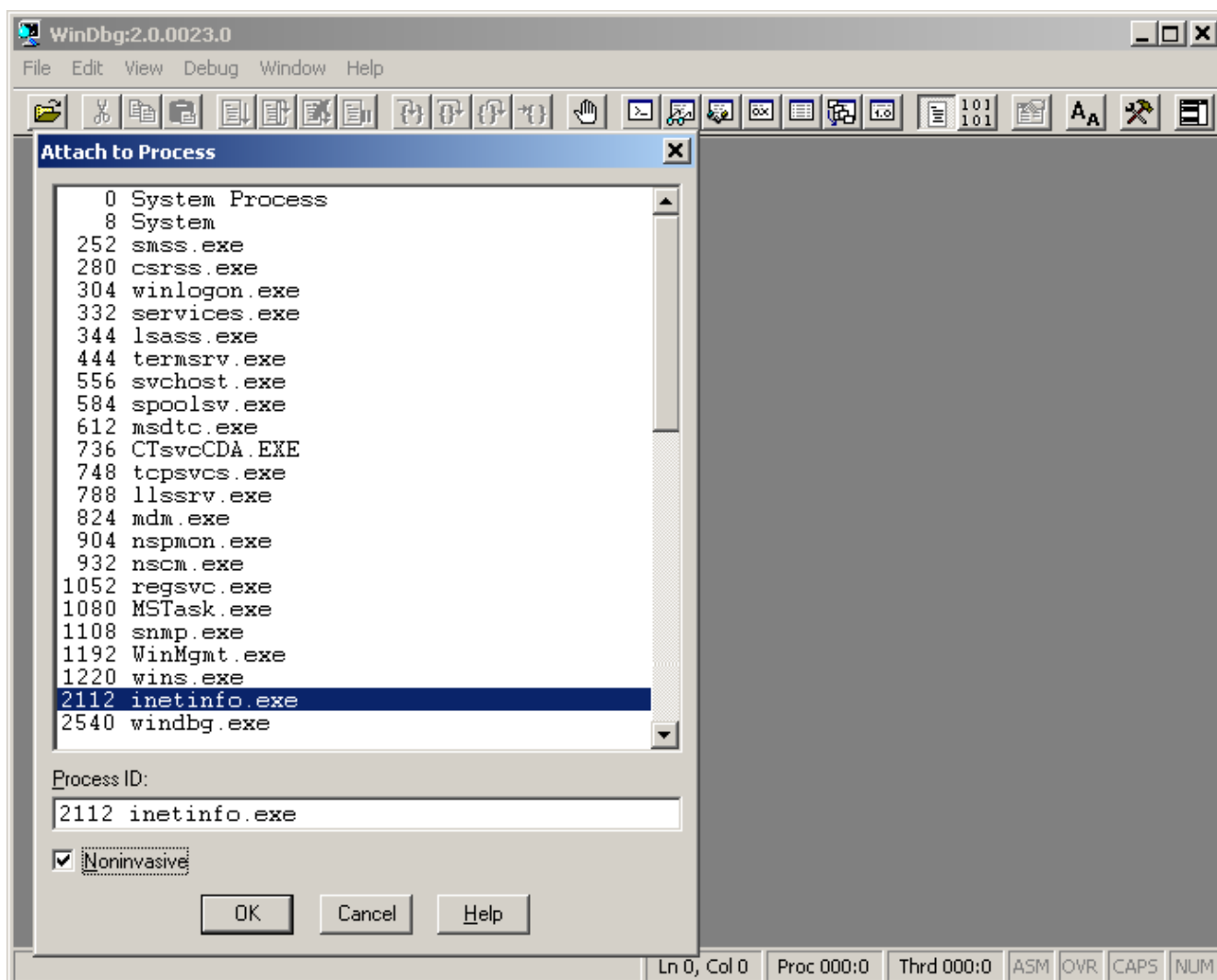


Figure 2-4: Use WinDbg to conduct a noninvasive audit of inetinfo.exe

Once WinDbg has attached noninvasively to inetinfo.exe you can examine process memory, thread call stacks, and conduct typical debugging session operations with the exception of controlling execution through commands like break, step, go, and the use of breakpoint conditions. Figure 2-5 shows WinDbg after it has attached noninvasively to inetinfo.exe. The Command window accepts debugger commands and each menu option you select that has a corresponding debugger command will display the command it issues on your behalf to enable you to type it manually into the command window in the future. WinDbg is useful for conducting numerous types of security audit and for diagnosing unexpected behaviors that might be indicative of the presence of malicious code. A few advanced uses are detailed in this book and this chapter shows its simplest auditing function: enabling you to view a complete list of the binary modules loaded into the inetinfo.exe process.

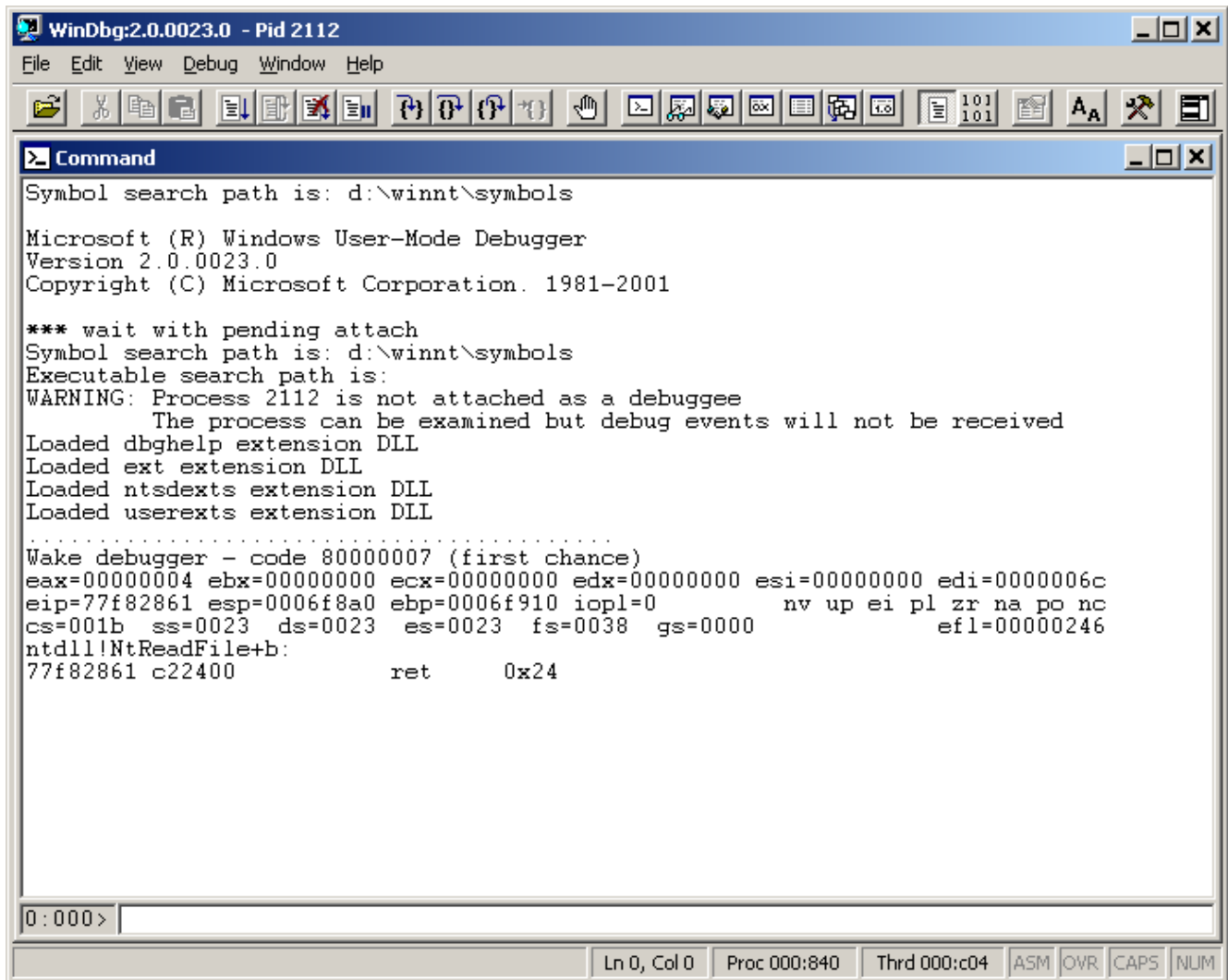


Figure 2-5: WinDbg noninvasive debugging lets you look but not touch

Awareness of the binary modules loaded into memory by inetinfo.exe is important for several reasons. Malicious code is easier to detect if you keep track of the origin and purpose of every binary module used by IIS. As in-process COM objects are loaded by dynamic content execution and as new script engines or ISAPI DLLs are activated the list of binary modules grows dynamically. Auditing the inetinfo.exe process regularly is an essential part of a comprehensive IIS security policy. Configuration changes, service pack and hotfix installations, or the addition of ancillary software that integrates with IIS will change the binary modules loaded by inetinfo.exe in ways that the makers of the software usually fail to document sufficiently. Documentation, even if it exists and looks complete, is no substitute for first-hand observation, regardless. You can spot potential security risks and investigate further by discussing binary modules with software vendors as soon as you see them loading into the IIS process. Ideally you conduct your own security review in a development environment before deploying any modifications to production servers and your audit of inetinfo.exe is a key part of that review. To see the complete list of binary modules loaded into the process to which WinDbg is attached choose Modules from the Debug menu. The Modules menu option brings up the Module List shown in Figure 2-6.

Name	Start	End	Timestamp			
inetinfo	01000000	01006000	Thu	Apr	26	13:42:56 2001
wamreg	65d60000	65d6e000	Fri	May	04	11:35:28 2001
svcext	671b0000	671bc000	Fri	May	04	11:35:21 2001
rpcref	68920000	68925000	Fri	May	04	11:35:13 2001
nsepm	69d00000	69d0d000	Fri	May	04	11:35:05 2001
metadata	6c7e0000	6c7f4000	Fri	May	04	11:34:53 2001
IisRTL	6e5a0000	6e5c1000	Tue	Oct	30	21:17:33 2001
IISMAP	6e5e0000	6e5f1000	Fri	May	04	11:34:44 2001
iisadmin	6e6f0000	6e6f7000	Fri	May	04	11:34:44 2001
SHLWAPI	70bd0000	70c34000	Fri	Aug	17	19:40:51 2001
COMCTL32	71780000	7180a000	Fri	Aug	17	19:40:55 2001
COADMIN	73330000	7333d000	Fri	May	04	11:34:30 2001
ADMWPROX	74e30000	74e3c000	Mon	Nov	29	23:31:11 1999
admexs	74e40000	74e4a000	Fri	May	04	11:34:22 2001
WS2HELP	75020000	75028000	Mon	Nov	29	23:31:09 1999
WS2_32	75030000	75043000	Fri	May	04	11:35:30 2001
WSOCK32	75050000	75058000	Fri	May	04	11:35:31 2001
SAMLIB	75150000	75160000	Fri	May	04	11:35:15 2001
NETAPI32	75170000	751bf000	Tue	Oct	30	21:19:36 2001
NETRAP	751c0000	751c6000	Mon	Nov	29	23:31:07 1999
Security	75500000	75504000	Mon	Nov	29	23:31:03 1999
IMM32	75e60000	75e7a000	Fri	May	04	11:34:17 2001
WMI	76110000	76114000	Tue	Nov	30	16:46:33 1999
MSASN1	77430000	77440000	Tue	Oct	30	21:12:09 2001
CRYPT32	77440000	774b5000	Fri	May	04	11:34:11 2001
CLBCATQ	775a0000	77625000	Tue	Oct	30	21:11:32 2001
WLDAP32	77950000	7797a000	Tue	Oct	30	21:11:51 2001
DNSAPI	77980000	779a4000	Tue	Oct	30	21:11:50 2001

Figure 2-6: Audit the list of binary modules loaded into inetinfo.exe using WinDbg

By keeping track of the binary modules upon which IIS depends you have a way to relate discoveries of security flaws in those binary modules to IIS security. In addition to dependencies on Win32 API binary modules, IIS depends on certain support modules provided by development tools used in its creation. The C runtime from Visual C++ is one such example. Table 2-3 lists each of the developer support modules upon which IIS are dependent. The C runtime module is listed in bold because the base IIS Admin service implemented by inetinfo.exe loads it also.

It is safe to assume that a patch or update to the C runtime from Visual C++ may have some impact on IIS, but you can't assume that Microsoft will immediately recommend its installation on systems that run IIS. When you learn of the release of such a patch and don't receive instructions to install it right away on production systems it is a very good idea to review the details of the fixes implemented by the patch and research what

impact, if any, the patch may have on IIS security from your perspective as a developer or administrator. If you see reason to be concerned you can coordinate with Microsoft Support and Microsoft Security groups to share findings.

Table 2-3: Developer Support Modules in inetinfo.exe v5.0.2195.2966

DLL Filename	Preferred Memory Range	Description
System32\msvcrt.dll	0x78000000 – 0x78046000	Microsoft C Runtime Library from Visual C++
System32\ATL.DLL	0x773E0000 – 0x773F2000	Visual C++ ATL Module for Windows NT (Unicode)
System32\mfc42u.dll	0x76FB0000 – 0x770A2000	Microsoft Foundation Classes ver 4.2 (Unicode)
System32\IMAGEHLP.DLL	0x77920000 – 0x77943000	Debug symbol engine
System32\vbajet32.dll	0x0F9A0000 – 0x0F9AB000	Visual Basic for Applications Development Environment - Expression Service Loader
System32\expsrv.dll	0x0F9C0000 – 0x0FA22000	Visual Basic for Applications Runtime - Expression Service

Win32 (or Win64) API binary modules and developer support modules are two of the five logical groups of modules that you will encounter as you perform your IIS audit. The other three groups are system modules, IIS modules, and third party application modules. Many of the modules loaded by inetinfo.exe come from the operating system to implement features including networking that aren't considered part of the base Win32 or Win64 APIs. These modules are listed in Table 2-4 with those modules that are part of the IIS Admin service listed again in bold.

Table 2-4: System and Networking DLLs in inetinfo.exe v5.0.2195.2966

DLL Filename	Preferred Memory Range	Description
System32\rpcrt4.dll	0x77D40000 – 0x77DB0000	Microsoft Remote Procedure Call Interface
System32\imm32.dll	0x75E60000 – 0x75E7A000	Windows 2000 IMM32 API Client DLL
System32\shlwapi.dll	0x70BD0000 – 0x70C34000	Shell Light-weight Utility Library
System32\wmi.dll	0x76110000 – 0x76114000	Windows Management Instrumentation DC and DP functionality
System32\clbcatq.dll	0x775A0000 – 0x77625000	COM Services
System32\msasn1.dll	0x77430000 – 0x77440000	ASN.1 Runtime APIs
System32\USERENV.DLL	0x77C10000 – 0x77C6E000	User environment and profiles
System32\dnsapi.dll	0x77980000 – 0x779A4000	DNS Client API DLL
System32\netrap.dll	0x751C0000 – 0x751C6000	Net Remote Admin Protocol DLL
System32\samlib.dll	0x75150000 – 0x75160000	SAM Library DLL
System32\WLDAP32.DLL	0x77950000 – 0x7797A000	Win32 LDAP API DLL
System32\lz32.dll	0x759B0000 – 0x759B6000	LZ Expand/Compress API DLL
System32\ntdsapi.dll	0x77BF0000 – 0x77C01000	NT5DS
System32\msafd.dll	0x74FD0000 – 0x74FEF000	Microsoft Windows Sockets 2.0 Service Provider
System32\wshtcpip.dll	0x75010000 – 0x75017000	Windows Sockets Helper DLL
System32\iphlpapi.dll	0x77340000 – 0x77353000	IP Helper API

System32\icmp.dll	0x77520000 – 0x77525000	ICMP DLL
System32\mprapi.dll	0x77320000 – 0x77337000	Windows NT MP Router Administration DLL
System32\activeds.dll	0x773B0000 – 0x773DE000	ADs Router Layer DLL
System32\adsldpc.dll	0x77380000 – 0x773A2000	ADs LDAP Provider C DLL
System32\rtutils.dll	0x77830000 – 0x7783E000	Routing Utilities
System32\SETUPAPI.DLL	0x77880000 – 0x7790D000	Windows Setup API
System32\RASAPI32.DLL	0x774E0000 – 0x77512000	Remote Access API
System32\RASMAN.DLL	0x774C0000 – 0x774D1000	Remote Access Connection Manager
System32\tapi32.dll	0x77530000 – 0x77552000	Microsoft Windows Telephony API Client DLL
System32\DHCPSCV.C.DLL	0x77360000 – 0x77379000	DHCP Server Service
System32\winrnr.dll	0x777E0000 – 0x777E8000	LDAP RnR Provider DLL
System32\irasadhlp.dll	0x777F0000 – 0x777F5000	Remote Access AutoDial Helper
System32\rsaenh.dll	0x01C60000 – 0x01C83000	Microsoft Enhanced Cryptographic Provider (US/Canada Only, Not for Export)
System32\rpcproxy\rpcproxy.dll	0x68930000 – 0x68938000	RPC PROXY DLL
System32\ntlsapi.dll	0x756E0000 – 0x756E5000	Microsoft License Server Interface DLL
Program Files\Common Files\SYSTEM\ole db\msdasql.dll	0x1F690000 – 0x1F6DA000	Microsoft Data Access - OLE DB Provider for ODBC Drivers
System32\msdart.dll	0x1F660000 – 0x1F67F000	Microsoft Data Access OLE DB Runtime Routines
Program Files\Common Files\SYSTEM\ole db\msdatl3.dll	0x01CD0000 – 0x01CE5000	Microsoft Data Access - OLE DB Implementation Support Routines
Program Files\Common Files\SYSTEM\ole db\msdasqlr.dll	0x1F6E0000 – 0x1F6E4000	Microsoft Data Access - OLE DB Provider for ODBC Drivers Resources
Program Files\Common Files\SYSTEM\ole db\oledb32.dll	0x1F8A0000 – 0x1F905000	Microsoft Data Access - OLE DB Core Services
Program Files\Common Files\SYSTEM\ole db\oledb32r.dll	0x1F910000 – 0x1F920000	Microsoft Data Access - OLE DB Core Services Resources
System32\odbcjt32.dll	0x01F70000 – 0x01FB2000	MDAC ODBC Desktop Driver Pack 3.5
System32\msjet40.dll	0x1B000000 – 0x1B16F000	Microsoft Jet Engine Library
System32\mswstr10.dll	0x1B5C0000 – 0x1B655000	Microsoft Jet Sort Library
System32\odbcji32.dll	0x027E0000 – 0x027EE000	MDAC ODBC Desktop Driver Pack 3.5
System32\msjter40.dll	0x1B2C0000 – 0x1B2CD000	Microsoft Jet Database Engine Error DLL
System32\msjint40.dll	0x1B2D0000 – 0x1B2F6000	Microsoft Jet Database Engine International DLL
System32\mtxdm.dll	0x6A790000 – 0x6A79D000	MTS COM Services
System32\comsvcs.dll	0x78740000 – 0x788A4000	COM Services
System32\txfaux.dll	0x6DE80000 – 0x6DEE3000	Support routines for TXF
System32\msdtpcx.dll	0x68C60000 – 0x68D0F000	MS DTC OLE Transactions interface proxy DLL
System32\mtxclu.dll	0x6A7A0000 – 0x6A7B0000	MS DTC amd MTS clustering support DLL
System32\CLUSAPI.DLL	0x73930000 – 0x73940000	Cluster API Library

System32\RESUTILS.DLL	0x689D0000 – 0x689DD000	Microsoft Cluster Resource Utility DLL
System32\msrd3x40.dll	0x1B270000 – 0x1B2BC000	Microsoft Red ISAM
System32\msjtes40.dll	0x1B7F0000 – 0x1B82A000	Microsoft Jet Expression Service

Each of the previous three tables listed modules that are often used in various programs created by developers and that are also used by inetinfo.exe. Table 2-5 lists the modules loaded into inetinfo.exe that were created specifically for use by IIS. The IIS modules include metabase.dll, the code that loads the metabase into memory and manages changes made to the metabase, standard network service modules that ship as part of the core set of information services provided by IIS such as ftpsvc2.dll, and iisadmin.dll which implements the core IIS Admin service. As in the previous three tables, modules loaded by inetinfo.exe for use by the IIS Admin service are bold.

Table 2-5: IIS DLLs in inetinfo.exe v5.0.2195.2966

DLL Filename	Preferred Memory Range	Description
System32\iisRtl.dll	0x6E5A0000 – 0x6E5C1000	IIS RunTime Library
System32\inetsrv\rpcpref.dll	0x68920000 – 0x68926000	Microsoft Internet Information Services RPC helper library
System32\inetsrv\IISADMIN.DLL	0x6E6F0000 – 0x6E6F7000	Metadata and Admin Service
System32\inetsrv\COADMIN.DLL	0x73330000 – 0x7333D000	IIS CoAdmin DLL
System32\admwprox.dll	0x74E30000 – 0x74E3C000	IIS Admin Com API Proxy dll
System32\inetsrv\nsepm.dll	0x69D00000 – 0x69D0D000	IIS NSEP mapping DLL
System32\iismap.dll	0x6E5E0000 – 0x6E5F1000	Microsoft IIS mapper
System32\inetsrv\metadata.dll	0x6C7E0000 – 0x6C7F4000	IIS MetaBase DLL
System32\inetsrv\wamreg.dll	0x65D60000 – 0x65D6E000	WAM Registration DLL
System32\inetsrv\admexs.dll	0x74E40000 – 0x74E4A000	IIS AdminEx sample DLL
System32\inetsrv\svcext.dll	0x671B0000 – 0x671BC000	Services IISAdmin Extension DLL
System32\inetsrv\W3SVC.DLL	0x65F00000 – 0x65F59000	WWW Service
System32\inetsrv\INFOCOMM.DLL	0x769B0000 – 0x769F2000	Microsoft Internet Information Services Helper library
System32\inetsrv\ISATQ.DLL	0x6D700000 – 0x6D712000	Asynchronous Thread Queue
System32\inetsrv\iisfecnv.dll	0x6E620000 – 0x6E625000	Microsoft FE Character Set Conversion Library
System32\inetsrv\FTPSVC2.DLL	0x6FC60000 – 0x6FC7F000	FTP Service
System32\inetsrv\smtpsvc.dll	0x67810000 – 0x67880000	SMTP Service
System32\fcachdll.dll	0x6FF20000 – 0x6FF2E000	IIS FCACHDLL
System32\rwnh.dll	0x68510000 – 0x68516000	IIS RWNH
System32\exstrace.dll	0x70120000 – 0x7012C000	IIS Async Trace DLL
System32\staxmem.dll	0x67390000 – 0x67396000	IIS Microsoft Exchange Server Memory Management
System32\inetsrv\ldapsvcx.dll	0x6CDB0000 – 0x6CDD2000	P&M LDAP Service
System32\inetsrv\anntpsvc.dll	0x69DB0000 – 0x69E4C000	NNTP Service
System32\inetsrv\isrpc.dll	0x6D660000 – 0x6D665000	ISRPC

File Name	Address Range	Category	Location
System32\inetsloc.dll protocol library	0x6E2B0000 – 0x6E2B8000	Internet	Service
System32\inetsrv\lonsint.dll	0x6CA80000 – 0x6CA86000	IIS NT specific library	
System32\inetsrv\ISCOMLOG.DLL Common Logging Interface DLL	0x6D6F0000 – 0x6D6FA000	Microsoft	IIS
System32\inetsrv\ladminx.dll	0x6CE30000 – 0x6CE44000	MP LDAP Server	
System32\inetsrv\storedbx.dll	0x67250000 – 0x67292000	MP LDAP Server	
System32\inetsrv\ldapaclx.dll DLL	0x6CE10000 – 0x6CE16000	U2 LDAP Access Control	
System32\inetsrv\seo.dll	0x681E0000 – 0x6821C000	Server Extension Objects DLL	
System32\inetsrv\sspifilt.dll Interface Filter	0x67400000 – 0x6740E000	Security Support Provider	
System32\inetsrv\compfilt.dll	0x732C0000 – 0x732C9000	Sample Filter DLL	
System32\inetsrv\aqueue.dll	0x74A60000 – 0x74AB0000	Aqueue DLL	
System32\inetsrv\gzip.dll	0x6FA20000 – 0x6FA2B000	GZIP	Compression DLL
System32\inetsrv\md5filt.dll	0x6C850000 – 0x6C85C000	Sample Filter DLL	
System32\inetsrv\httpext.dll Windows NT	0x6EEB0000 – 0x6EEF2000	HTTP	Extensions for
System32\inetsrv\dscomobx.dll	0x71CF0000 – 0x71D22000	MP LDAP Server	
System32\inetsrv\ldapdbx.dll	0x6CDE0000 – 0x6CDF8000	ILS Service helper dll	
System32\inetsrv\iilsdbx.dll	0x6E500000 – 0x6E511000	MP LDAP Server	
System32\inetsrv\IISLOG.DLL	0x6E600000 – 0x6E615000	Microsoft IIS Plugin DLL	
System32\inetsrv\ntfsdrv.dll DLL	0x69C20000 – 0x69C2C000	NTFS	Message Store
System32\inetsrv\mailmsg.dll DLL	0x6C950000 – 0x6C963000	Mail	Message Objects
System32\inetsrv\nntpfs.dll DLL	0x6A160000 – 0x6A183000	NNTF File System Store Driver	
System32\query.dll	0x0CA70000 – 0x0CBD0000	Content Index Utility	
System32\inetsrv\wam.dll DLL	0x65D80000 – 0x65D95000	Microsoft Internet Server WAM	
System32\inetsrv\wamps.dll	0x65D70000 – 0x65D76000	WAM Proxy Stub	
System32\inetsrv\iwrps.dll	0x6D5F0000 – 0x6D5F7000	IWamRequest Proxy Stub	

Conduct a regular audit of inetinfo.exe and the out of process hosts that run on your server. The binary modules that get loaded into out of process host processes (mtx.exe, dllhost.exe, or w3wp.exe) can also be audited using WinDbg noninvasive debugging. Attach noninvasively to a process just long enough to view its list of modules, using screen capture or another technique to copy the list of modules displayed by WinDbg, then choose Stop Debugging to release the process and restart its threads. You can conduct noninvasive debugging at any time, but doing so during off-peak times will minimize processing delay for users currently connected to your hosted applications.

Programmable Extensibility

One of the most valuable features of the IIS platform is the extent to which custom applications can be built that extend the platform's functionality. The extensibility of IIS goes far beyond the existence of Active Server Pages for server-side scripting. IIS provide object-oriented interfaces for programming and configuration with Active

Directory Services Interface (ADSI) for manipulating the metabase, Internet Server API for creating server-side plug-in extensions and filters, native platform support for COM and COM+, and integration with the Microsoft .NET Framework.

Active Template Library ISAPI Extensions and Filters

The Internet Server Application Programming Interface, ISAPI, enables the creation of server-side plug-ins for creating content handlers that are invoked whenever a client requests a file hosted by IIS with a particular file extension. ISAPI also enables plug-in filters that layer automatically into request processing in an event-driven manner. Any programming language that can be used to create Windows DLLs can be used to build ISAPI extensions and filters. Microsoft also provides class libraries for building common types of ISAPI module more quickly and easily. A part of the Microsoft Foundation Classes class library wraps lower-level ISAPI calls with five classes: CHttpServer, CHttpServerContext, CHtmlStream, CHttpFilter, and CHttpFilterContext. Using these MFC classes you can build ISAPI DLLs that implement extension functions for content handler modules, filters to layer in custom request processing logic for all content types, or both as a single module.

The other class library Microsoft provides is specifically for C++ and it is much more powerful and comprehensive than the MFC ISAPI classes. Called the Active Template Library (ATL) and applicable for building high-performance low memory overhead COM objects as well as ATL Server modules, this library for C++ is the most powerful development tool for ISAPI. Using the ATL Server library developers can create XML Web services using SOAP and most of the other services enabled by the .NET Framework using a lower-level SDK that gives fine-grained control over everything from memory management to cache management, thread pooling, and security. ATL Servers are built by more experienced developers who are familiar with C++, COM, DLLs, network protocols, low-level memory management, and creating secure code.

Integration with .NET

The Microsoft .NET Framework creates a platform for the development of managed code and managed information systems that bring security right to the core of every decision made by every line of code. Code Access Security and role-based security, which includes low-level evaluation of a user's security context established as the minimum permissions required for execution of code by a user or a member of a particular group, that is compatible with COM+ services' own role-based security provisions allows for distributed object-oriented applications that implement unprecedented security. In addition to these and other .NET Framework security fundamentals, the new version of Active Server Pages brings the power of the .NET Common Language Runtime and the Microsoft Virtual Machine to the IIS development platform with enhancements designed to eliminate security vulnerabilities that plagued Internet application development in the past.

ASP.NET

IIS 5 and 6 implement classic Active Server Pages as well as an enhanced Active Server Pages for .NET called ASP.NET. Like IIS 6 themselves, ASP.NET changes the architecture inherited from the foundation of its predecessor to support multiple worker

processes hosted in `aspnet_wp.exe` and a new ISAPI Extension that invokes ASP.NET processing for applications hosted by IIS. In addition to standard Web applications built around .NET, ASP.NET enables IIS to be used as a platform for building and deploying XML Web services.

.NET Passport

Microsoft .NET Passport is a single sign-on service that enables third-party Web sites to authenticate users against the .NET Passport user database. All of the best practices for securing user credentials and protecting user privacy are implemented by .NET Passport including the use of SSL encryption to protect user credentials from interception by eavesdroppers. IIS integrates the .NET Passport authentication service by way of an ISAPI filter named `msppfltr.dll` that is installed by default in the `system32\MicrosoftPassport` directory.

Integration with COM+

COM does not preserve user security account identity tokens set at the level of individual threads within a process when one of those threads makes an interprocess call to another COM object. COM only preserves the user account token that is set for the host process when making calls to out of process COM objects. For IIS 4 and 5 this causes serious security problems for any in-process Web application hosted inside `inetinfo.exe` because the `inetinfo.exe` process typically runs as the local System account. Any in-process application that makes use of COM is able to execute code out-of-process using the System user account security context with the System token. Any malicious code deployed to an IIS 4 or IIS 5 server box is able to take complete control of the box by instantiating non-malicious out of process COM objects inside in-process Web application code when those non-malicious COM objects provide access to system resources and rely upon the security token of the caller to enforce security policy.

IIS 4 attempts to resolve this problem by cacheing an in-process thread's security token whenever an out of process MTS component is created. Using an undocumented behind-the-scenes mechanism, IIS 4 transfers the cached token to MTS so that it can replace its own process token and reimpersonate the user security context of the calling remote thread. This accomplishes the minimum functionality required for integrating MTS packages into Web applications but this work-around doesn't stand up well to properly designed malicious code. It also creates a dependency on the Single Threaded Apartment model for deploying COM objects, which is less than optimal on a busy server. The result is a technical reality that IIS 4 is especially vulnerable to malicious code deployed to it by authorized active content publishers. Hosting all content out of process on IIS 4 is therefore mandatory so that hosted code's security token is that of `IWAM_MachineName` even in the worst-case scenario. A properly locked-down and hot fixed IIS 4 server can host multiple Web sites with minimal risk, but only by disabling all active content and hosting third party Web sites out of process. Allowing third-parties to deploy code to an IIS 4 based server is not a supported usage scenario.

IIS 5 attempts to resolve this problem by using COM+ instead of COM. COM+ was first deployed on Windows 2000 and it supports a new interprocess impersonation mode called cloaking to directly resolve COM impersonation limitations. IIS 5 also supports hosting out of process in a single application pool, one process that hosts multiple Web

applications, as an alternative to hosting every out of process application in a separate process as was necessary in IIS 4. However, exploits are still possible from inside any in-process application, and any out of process application that is configured to live in the single pool can mount attacks against the other applications in the pool. A properly locked-down and hot fixed IIS 5 server can host third-party Web sites with active content with minimal risk. In-process applications that do not include active content are also relatively safe under IIS 5, even for hosting third-party Web sites as long as only GET and HEAD HTTP requests are allowed. Still, the possibility that buffer overflow vulnerabilities may exist that have gone undetected should make you nervous enough to avoid in-process applications under IIS 5 in spite of the small performance benefit they provide. The small benefit does not outweigh the risk that a buffer overflow attack may give control of the server box to an intruder.

IIS 6 throws out the misplaced notion of in-process applications completely. There never should have been any such thing, and reports by the media that took Bill Gates' statements out of context said, to paraphrase, that he was very sorry in-process Web applications had ever been created as part of IIS and Microsoft would take immediate action company-wide to deploy an improved platform for trustworthy computing. IIS 6 also supports COM+ 1.5 which enables the following new features:

- Interface Fusion: binding to explicit objects and DLLs based on version
- Service Domain Partitioning: decoupling COM+ contexts from objects
- Thread Pool Apartment Selection: selecting threading model for active content
- COM+ Tracker: debugging with native COM+ tools

The integrated support for access to COM+ services by way of automatic configuration performed by IIS Admin when new applications are configured under IIS and by way of server API support for instantiating any COM or COM+ object that is registered on the server provides powerful application development benefits. These features also expose a great deal of security risk, and throughout this book you will see how the integration with COM and COM+ can be turned into security exploits under the right conditions. One of the ongoing challenges of securing and keeping secure any IIS installation is its inseparable integration with COM+.

Nuclear and Extended Family

Internet Information Services form the foundation for numerous ancillary products created by Microsoft as well as third-party independent software vendors. These optional add-on products collectively represent the IIS extended family of products. The nuclear family is comprised of the integrated e-mail, network news, and File Transfer Protocol servers as well as features that are tightly integrated and available as default or optional installations along with IIS Web publishing services. These include, as previously discussed, the FrontPage and Office Server Extensions, Active Server Pages, and Remote Administration features in addition to WebDAV for remote access to publishing points, Index Server, Internet Locator Server, Proxy Server, and the Microsoft Message Queuing HTTP IIS extension to name but a few.

Nuclear SMTP, NNTP, FTP, and Publishing Point Services

Simple Mail Transfer Protocol (SMTP) is the Internet standard for e-mail relay defined in RFC 2821. SMTP servers listen by default on port 25 for inbound TCP connections from SMTP clients, which can be either e-mail programs operated by end users for the purpose of sending e-mail messages or mail relay hosts, normally SMTP servers themselves, that store and forward messages until they arrive at the mailbox of the intended recipient by way of mail exchangers configured for the destination domain. IIS include a feature-rich and industry-leading secure SMTP server with support for several types of client authentication and relay session encryption that is loaded into inetinfo.exe through smtpsvc.dll.

RFC 2821 (SMTP) is online at <http://www.ietf.org/rfc/rfc2821.txt>

Network News Transfer Protocol (NNTP) is designed to facilitate distributed discussion groups that can optionally be replicated automatically between NNTP servers. The protocol, documented in RFC 977, defines a system for reading, posting, and managing large message databases on an NNTP server that serves messages to NNTP clients. NNTP clients can be newsreader programs operated by end users or other NNTP servers to which messages are relayed. IIS versions 4 and 5 include a built-in NNTP server loaded into the inetinfo.exe process through nntpsvc.dll.

RFC 977 (NNTP) is online at <http://www.ietf.org/rfc/rfc0977.txt>

File Transfer Protocol (FTP) is a session-oriented password-protected file server system where clients login to the FTP server to send and receive files. The protocol specification includes a facility for navigating directory hierarchies on the server and selecting files to retrieve or folders into which to send files. IIS include an FTP server that conforms to the FTP specification as documented in RFC 959.

RFC 959 (FTP) is online at <http://www.ietf.org/rfc/rfc0959.txt>

IIS also include several mechanisms to enable publishing to the servers managed by IIS Admin. Each distinct location to which content can be published from remote locations on the network is termed a publishing point, and access to the publishing points established on an IIS box is controlled using technologies including FrontPage Server Extensions, WebDAV, File Transfer Protocol, and BITS in addition to any application-specific provisions for remote management of publishing points.

Products That Extend The IIS Foundation

Several important add-on products provided by Microsoft exist by virtue of the way that IIS was built as a platform for developing TCP/IP network services. Among these products, which are each part of the IIS extended family, are Index Server, Internet Locator Server, Microsoft Proxy Server, the Microsoft Message Queuing HTTP IIS extension, Site Server with its Personalization & Membership facility including Direct Mail and Site Server Analysis, Site Server Commerce Edition, the Microsoft Commercial Internet System, Application Center, Commerce Server, and the XML nirvana for business objects: BizTalk Server. This book provides you with the technical foundation to secure each of these products effectively even if each one is not detailed to the same degree as the others because they all share the IIS foundation.

A solid understanding of the IIS architecture that includes low-level awareness of its component parts and the security risks inherent to each of them is necessary to properly secure and monitor the operation of this rich service development platform. Whether you're a developer or an administrator, security for IIS is an essential element of your technical design as you engineer and support production information systems for the Internet built using Windows servers.

Chapter 3: Server Farms

Few IIS deployments exist as isolated server islands. The goal of isolating a server is to achieve provable security by removing variables and eliminating threats. This is accomplished by building self-contained servers that have no relationship to or interaction with other servers on the network. A server island, for lack of a better term, offers a practical security advantage because it can be configured to explicitly reject and ignore everything except client requests for the service it provides through a single TCP port.

This service can be addressed using only a single IP address, with no DNS naming that can be hijacked or spoofed, and an SSL certificate can be issued by a certificate authority (CA) that binds a public key to the IP address as the common name of the box. Additionally, such a box can be physically connected directly to a router rather than an intermediary network hub or switch. On the surface such a box appears to have only the following four points of potential vulnerability:

1. Security bugs in the services provided by the box
2. Security bugs in the network interface device drivers that enable the box to communicate with the network
3. IP spoofing and router hijacking between the server island and its clients
4. A breakdown in the chain of trust relied upon by the client to allow it to reliably authenticate the server through its IP address common name

The last two vulnerabilities are outside of your control unless you issue your own certificates (see Chapter 14) and develop your own thick client applications (see Chapter 10). However, the first two vulnerabilities can be managed and monitored locally at the box using a variety of hardware and software tools, a comprehensive security policy, tight controls on the physical security of the box, and a secure procedure for deploying new code and new content to the box. But a server island is not the security solution of choice for most Web applications. As isolated singular network nodes, server islands can be knocked offline by a relatively simple DoS attack. Even a non-malicious usage flood will always overwhelm the box as it reaches its processing capacity and bandwidth limits. A single hardware failure causes downtime that impacts all users. If the box is compromised by an intruder, all is lost. The very thing that makes a server island more secure also makes it more fragile. An operating system clustering service that enables failover N+1 configurations, or N (active) installations with 1 (spare) installations of the same OS on identical equipment functioning together as a single clustered node, can reduce downtime risk for a single node due to hardware failure by providing automatic failover redundancy.

But this type of failover clustering is costly because it duplicates hardware without duplicating capacity. An intruder who takes control of such a single-node cluster owns the entire node anyway because the clustering service automatically mirrors any malicious code sent to the node by the attacker so that malicious code keeps running, too, in the event of a hardware failure in one part of the cluster. DoS attacks against this type of cluster are also no more difficult than DoS attacks against network nodes that consist of single server islands because the network, rather than the server cluster's aggregate

processing speed, determines the throughput bottleneck and the saturation point at which DoS occurs. Or, DoS security bugs that exist in the same way in each component of the clustered node are all exploited to knock out the entire cluster just as they would be exploited to knock out the node if it consisted only of a single server island that was also subject to the DoS security bug.

The preferred solution for reliability and security risk management for Web servers that increases capacity as new hardware is added and eliminates the single point of network failure vulnerability of a server island or a single-node cluster is to build and deploy a network load balancing (NLB) server farm. A server farm that implements NLB is a cluster of cooperating network nodes that are secured individually, managed together as a single unit, and configured to automatically balance the workload of client requests and service processing while containing security incidents to individual nodes that can be removed from the farm before they impact its overall health. The source of an attack that compromises the health of one node can be blocked immediately so that it doesn't impact the other nodes in the farm while they are hardened against the successful attack. Server farms can be distributed geographically and grown dynamically to service larger loads.

They are also more resilient as nodes that come under DoS attack are removed from service temporarily, forcing such an attack to target the entire farm or its aggregate bandwidth through a Distributed Denial of Service (DDoS) attack in order to achieve a DoS condition. NLB clusters may incorporate hardware clustering rather than stand-alone server boxes at any or all NLB nodes. Securing an IIS server farm requires special precautionary countermeasures due to the increased complexity of distributed management, service configuration, provisioning, and security monitoring.

Understanding Farms and Gardens

There are many ways to build and manage server farms. Some developers and administrators prefer to pay a premium for somebody else to anticipate and solve all of the problems associated with configuring and managing server farms. There's nothing wrong with a specialized product suite to enable your server farm, except that security vulnerabilities discovered in the product by malicious attackers who take advantage of somebody else's deployment of the same product suite (or their own deployment; some malicious attackers have money to spend or manage to steal or defraud to get their hands on security products) translate directly into vulnerabilities for your farm. However, the security and management problems for a server farm aren't so complicated that you should avoid solving them yourself with custom code. If there's one place that custom security code is always appropriate, and even preferred over standardized well-known security code, it's in the creation of application logic to implement your business-specific specialized security policies.

Managing security for a server farm means deploying tools to monitor activity and implement security policies and procedures consistently throughout the farm. If your sever farm is designed to host applications on behalf of other people, or host many distinct unrelated applications on your own behalf, then the farm is said to consist of multiple gardens. A garden is typically walled off from other gardens with process boundaries on the IIS box. Ideally, a Web garden is configured to use a pool of processes, any of which can service requests sent to the application just as in a server farm the nodes in the farm are pooled

and any node can service client requests in exactly the same way as any other node. The use of single-process application pools where multiple applications share the same process space also qualify as Web gardens where each process represents one garden rather than garden boundaries being marked by the boundaries of each application and each process pool. The precise semantics aren't too important. What is important is that a garden that isolates an application from all other applications with a process boundary is more secure because code executing on the threads of the host process can't destabilize code running on behalf of other applications on other threads. Further, an application that lives in a process pool is more reliable because it has backup processes that share the load within the node and keep the application alive even if one process in the pool blows up. Figure 3-1 depicts the difference between a server farm that consists of load balancing application gardens and one that consists of isolated server islands.

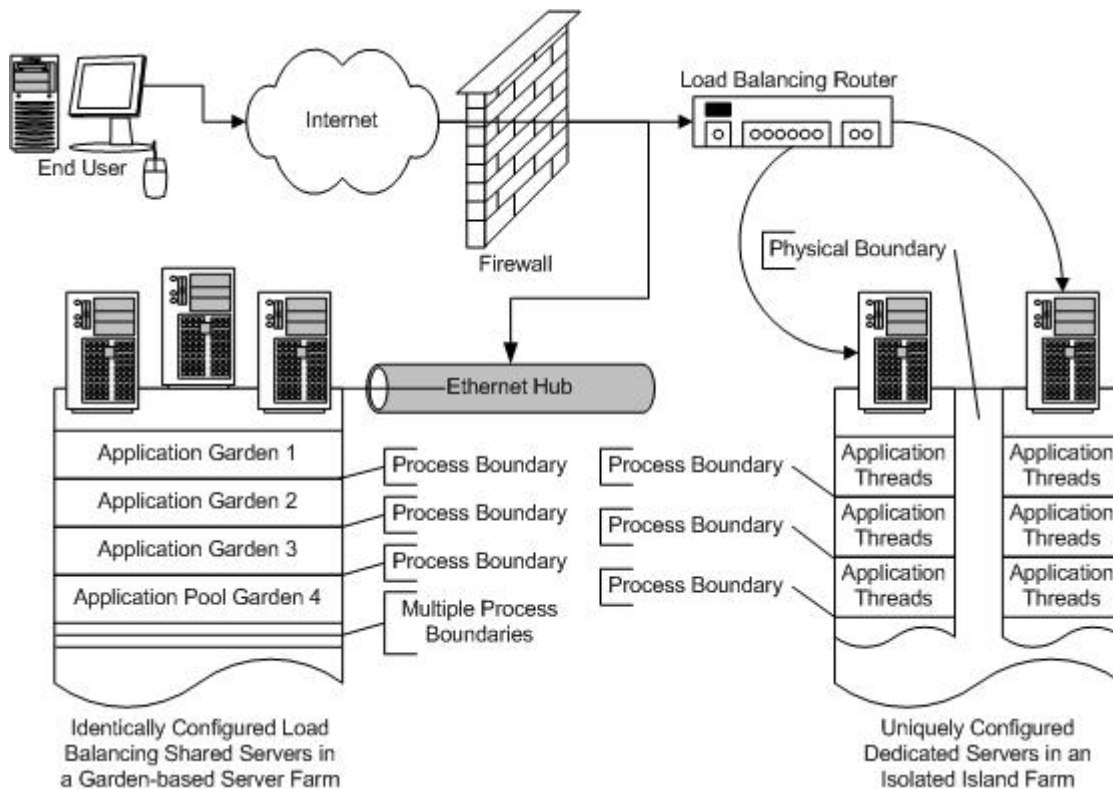


Figure 3-1: Load Balancing in a Server Farm with Gardens Compared to Server Islands

The defining characteristic of a garden is the planting of multiple unrelated applications that attract distinct user communities where the applications and content are managed and developed by different people. Each application is tended to separately and each application's gardener need not be concerned (much) with what other gardeners do in their respective neighboring gardens. This implies access to shared resources such as hard disk space, processor time, network bandwidth, and physical security management in much the same way as sunlight, water, air, and police protection exist as shared resources upon which physical gardening depends. The shared resource is either apportioned to the individual applications by thread scheduling in the pool or by process scheduling in an isolated process if one is established for and dedicated to an application. Another term commonly used to describe a garden, shared server, is descriptive and useful for marketing but since the larger deployment of server boxes that

replicate content and load balance request processing is referred to as a server farm the gardening metaphor makes sense and it distinguishes a shared server island from a proper shared server farm consisting of process boundary-isolated NLB gardens balanced across all NLB nodes of the farm.

Load Balancing Proxies and Switches

A server farm begins with a method for balancing load. One of the original methods is called round robin DNS in which a particular FQDN is mapped to a pool of IP addresses. The DNS servers that supply authoritative nameservice for the domain rotate through the address pool so that different client resolvers are supplied with different addresses and requests are therefore directed at different servers in the server farm in a manner that is transparent to the end user. However, this type of load balancing doesn't work well for preventing DoS conditions due to the fact that certain IP addresses of servers in the farm may become unreachable due to attacks or hardware failures, and since the DNS will continue to hand out those temporarily inaccessible IP addresses for a period of time (the Time To Live, or TTL, configured for the zone in DNS, at a minimum) until DNS cache expires at both the client resolver and the DNS server queried by the client resolver. There is no way to ensure continuous service to all clients. DNS round robin load balancing serves a useful purpose only if you start with a pool of IP addresses that point to mirrored server farms where additional load balancing takes place and high availability can be assured for each farm. Figure 3-2 shows the DNS round robin architecture.

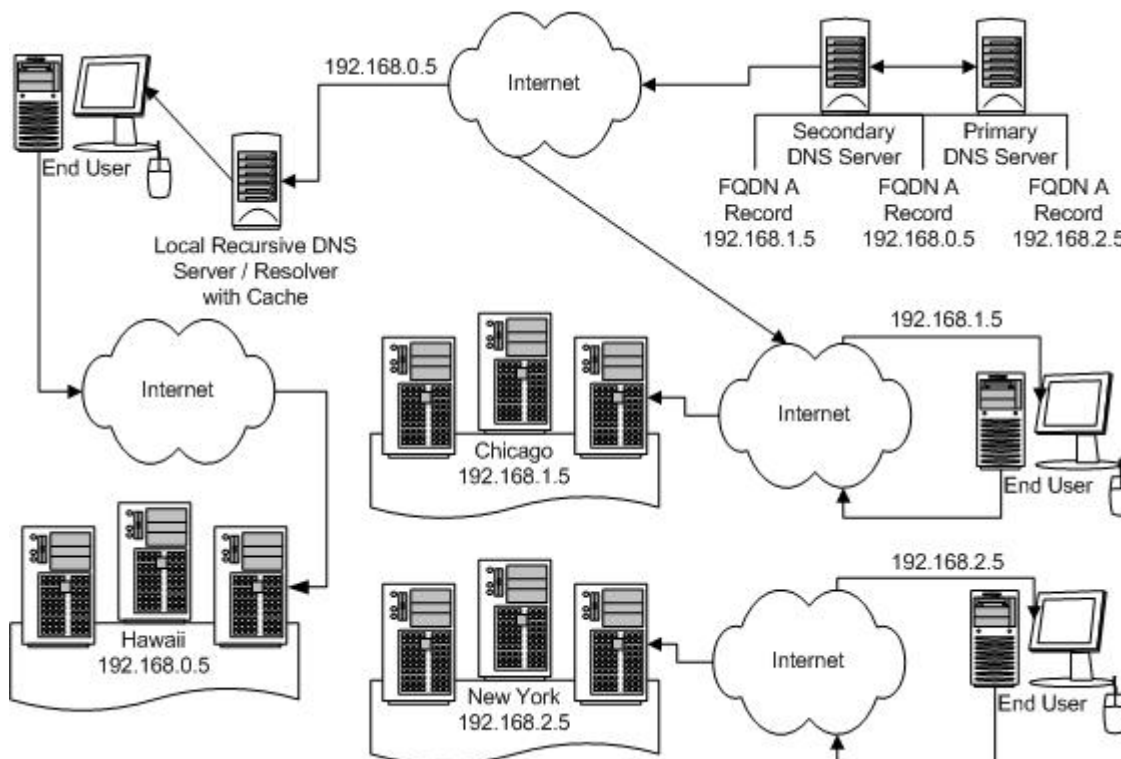


Figure 3-2: DNS Round Robin Load Balancing Between Farms

If your server farm is not distributed across multiple redundant locations then round robin may not be appropriate for load balancing. When it is used, your ISP must be capable of failover routing between each farm, and each server in each farm must be multiple

homed (addressed with each IP address assigned to all farms) in the event of hardware failure or other disaster that impacts the health of particular farms in the distributed cooperating collection of redundant farms. Otherwise round robin load balancing is useful only to mitigate damage and disruption when one or more farms is destroyed or DoS'ed. By setting the TTL on your domain's zone SOA (Start of Authority) record in a round robin load balancing authoritative DNS server to one hour, you place a cap on the amount of time that a portion of the network will be unable to access your servers during an incident that impacts some but not all of your redundant farms while preserving access from any client whose DNS resolver has cached an IP address that points to a farm that is still accessible. By promptly removing affected addresses from the round robin address pool when an incident or outage occurs you prevent new DNS lookups from resulting in inaccessible IP addresses if your ISP can't failover route to one of your other farm locations. This contains the damage, but doesn't eliminate it completely. Whether or not round robin DNS is employed for the domains hosted by your server farm, the TTL should be set low, at an hour or less, because you may need to relocate your equipment to new IP addresses as part of disaster recovery and a long TTL increases the delay to bring services back into operation when replacement IP addresses are configured in authoritative DNS servers for each of the domains hosted in your relocated farm.

Another network load balancing solution is called a reverse proxy. A reverse proxy acts as a single point of contact for all incoming requests and implements NLB to select a server in the farm to which to relay the request based on a load balancing algorithm that may include feedback about the current processing load of each server in the farm. The reverse proxy keeps the TCP connection open with the HTTP client while it relays the client's request to and receives a response from the selected server. The reverse proxy then delivers the response to the client. Depending upon configuration settings or the design of the reverse proxy, the response may be buffered in its entirety by the reverse proxy before data is sent back to the client so that the reverse proxy can try sending the client's request to a different server in the farm if the server it contacted initially fails during request processing. Or the reverse proxy will relay response data to the client as it is received from the server and let the client do what it can with whatever data it receives up to the point of a request processing failure. Typically, in a case such as this, the client re-sends the request when the user sees the incomplete response and presses the refresh button, starting the whole sequence of events over again. Figure 3-3 shows the reverse proxy load balancing architecture. This architecture has certain advantages, but it can complicate transaction processing in sensitive Web applications that rely on the TCP network protocol to provide endpoint-to-endpoint data communications integrity because the reverse proxy injects another pair of endpoints into the HTTP request processing sequence. TCP guarantees endpoint-to-endpoint link integrity and verifies that data sent by one endpoint is received at the other endpoint but it wasn't designed to manage chained endpoint pairs. The reverse proxy, which itself may in turn be talking to a client proxy rather than a client endpoint, is potentially a weak link in the reliable and timely notification to the server of success or failure at the final TCP endpoint.

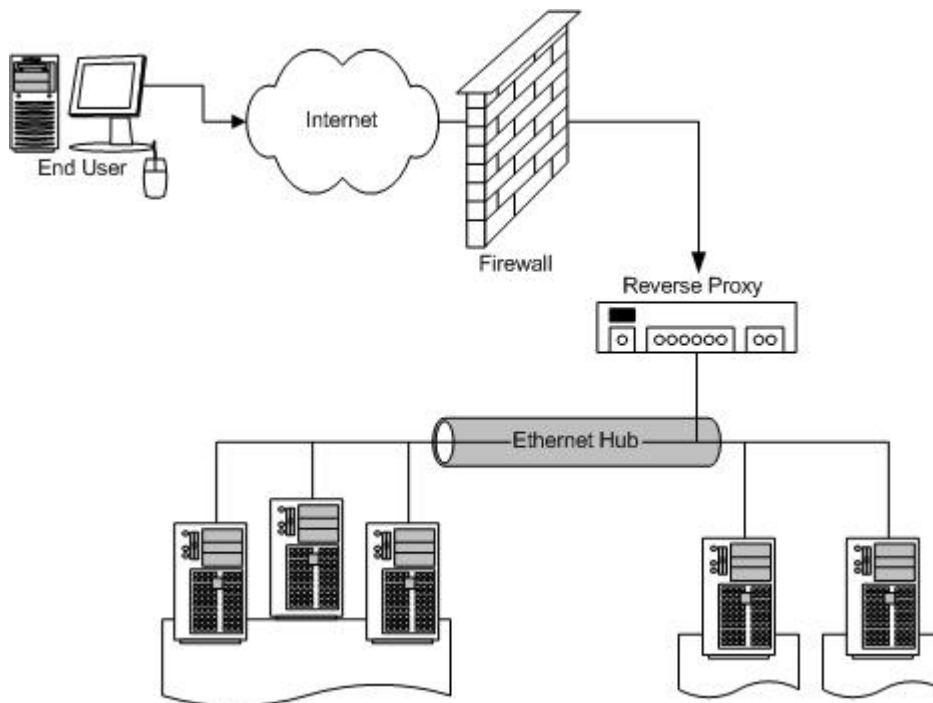


Figure 3-3: Load Balancing Using a Reverse Proxy Server

The value of a reverse proxy is in the application-specific knowledge it can embody that lets load balancing occur in a way that is just right for your application design and server farm configuration decisions. For example, it can load balance client requests to particular NLB nodes based on a particular HTTP cookie header, or URL-encoded name/value pair, or even the unique identifier used in an SSL session that is capable of session reuse or session caching. A reverse proxy may serve as a high-capacity high-throughput hardware-accelerated SSL gateway and communicate with a particular node in the server farm only for application request processing. Or the reverse proxy can simply relay SSL-encrypted data on to an NLB node that is itself capable of SSL. In the former scenario only the high-performance reverse proxy needs to have an SSL certificate installed, and the reverse proxy must be protected by a firewall, and in the latter scenario each NLB node must have a copy of the same SSL certificate installed so that each one is able to authenticate correctly using the same common name, or FQDN, when receiving SSL-secured client connections.

A load balancing switch is a special device that implements either reverse proxy load balancing or Network Address Translation (NAT) load balancing and routing that rewrites the destination IP address of incoming packets so that a node in the server farm receives all packets that pertain to a particular TCP connection. NAT also rewrites the source IP address of outgoing packets sent by the node to the client so that the IP address of the router appears in the source address of each packet that will be routed to another network over the Internet. Both reverse proxies and NAT routers can work with UDP packets as well, although load balancing of UDP traffic is quite a bit different than TCP traffic and IIS doesn't support any UDP-based application protocols so you don't need to worry about it, most likely. Load balancing switches provide context-sensitive and application-aware load balancing that is geared toward solving some of the problems of session state management that arise out of the stateless nature of HTTP client/server interactions. When your application development efforts do not need or want to tackle the

issue of session management at the Web application level, a load balancing switch can solve the session management problem for you in hardware, making your application development a little easier.

Windows Network Load Balancing

The Microsoft Windows Network Load Balancing Service provides an alternative load balancing solution for IIS clusters that run Windows NT 4 Enterprise Server (with the Load Balancing Services optional add-on component), Windows 2000 Server, or Windows .NET Server operating systems. Windows NLB configures two IP addresses per cluster node, one that is unique to the node and is used for communications between nodes or between a node and other servers, such as database servers, on the network, and one that is the same for all nodes in the cluster. The Windows NLB device driver allows each cluster node to share the same IP address, a configuration that would result in an error without the NLB device driver. The shared address is referred to as the primary IP address and the unique address is referred to as the dedicated IP address. What the shared addressing enables is a load balancing solution without a load balancer. Figure 3-4 shows the architecture of a Windows NLB server farm.

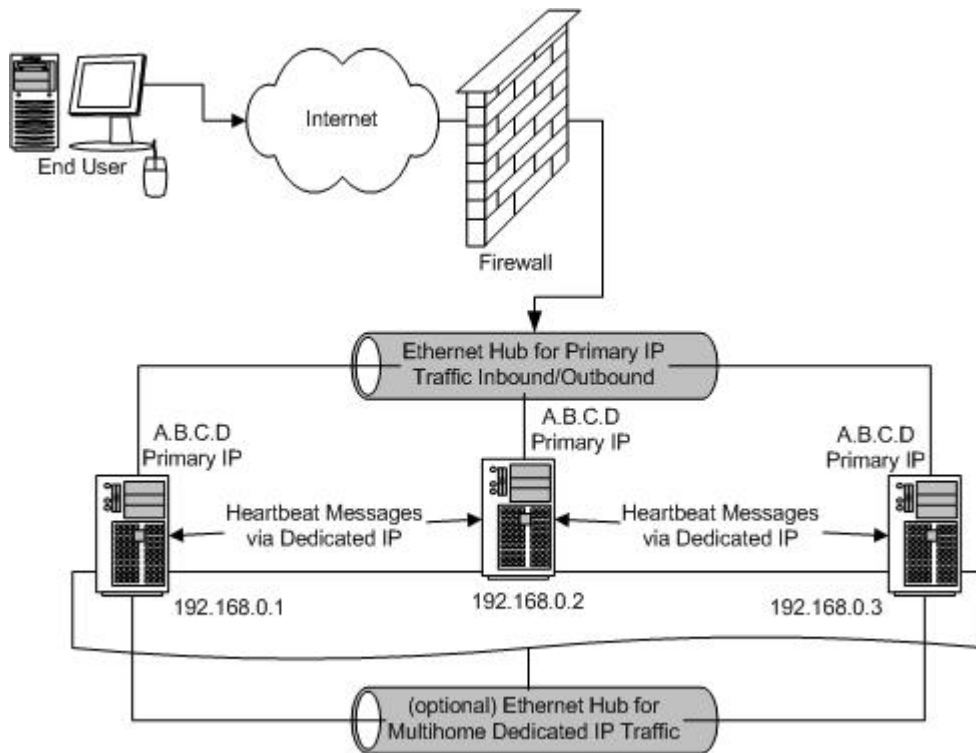


Figure 3-4: The Windows Network Load Balancing Server Farm Architecture

With Windows NLB there is no need for a reverse proxy or NAT router. Each cluster node is simply connected to the LAN so that each node receives all packets addressed to the primary IP address and the nodes each decide which TCP connections to service, and therefore which IP packets to process and which to ignore, based on a simple load balancing algorithm that takes into consideration the number of nodes in the cluster and a few load balancing configuration settings that are set the same on each cluster node. The nodes in the NLB cluster exchange heartbeat messages using their unique dedicated IP

addresses, which you can bind to a second network adapter in each node that connects to a second isolated LAN if you wish to separate cluster internal traffic from request processing traffic. When nodes fail, or are removed from the cluster, the remaining nodes carry out a reapportionment procedure known as convergence where they determine how many nodes are left in the cluster and therefore how to divide up request processing load. Nodes can be set up to share load actively or monitor passively and wait for another node to fail before activating to provide automatic failover service.

LAN/WAN Security and Firewalls

Network security is an important part of protecting a server farm and any installation of IIS, although the topic is not the specific focus of this book. Aside from being aware of the risks and configuring network hardware and software to their most secure settings that are still compatible with your objective of servicing requests from users, and of course preserving the physical security of equipment connected to the network, your primary network security responsibilities as an IIS administrator or programmer are to anticipate problems and avoid causing any new ones. It is up to you to implement additional security countermeasures within IIS or its hosted applications in order to compensate for network security holes, either known or unknown, that might impact the server farm. A firewall is typically a cornerstone of network security, yet it usually does nothing more than filter packets based on port number. You need more capabilities than just port number filtering, and those capabilities need to include tools to enable a dynamic response based on the full context of incidents as they occur. Ideally, security information gathered by all network devices would be integrated to provide a rich context against which the authenticity and validity of all network traffic is automatically evaluated. If automatic actions aren't taken in response to this information, warning signs of malicious activity can at least be written to logs or sent directly to humans who will research the activity further to decide whether it compromised transactions or systems in the farm.

Intrusion Detection Systems

It's one thing to firewall your server farm so that TCP/IP traffic is only allowed to reach your equipment if it contains an authorized destination port, such as 80, and application protocol, such as HTTP. Firewalls are useful tools for data security. However, it's another thing entirely to detect intruders' malicious TCP/IP traffic and automate dynamic countermeasures to protect your equipment from attacks. Blocking packets that contain crucial counter intelligence about attackers' malicious TCP/IP traffic throws the baby out with the bathwater. What you really want is an intrusion detection system (IDS) that feeds back security information about IP addresses that are responsible for suspicious behavior into Web application code that enforces security policy. Figure 3-5 shows the architecture of a typical Intrusion Detection System protected server farm.

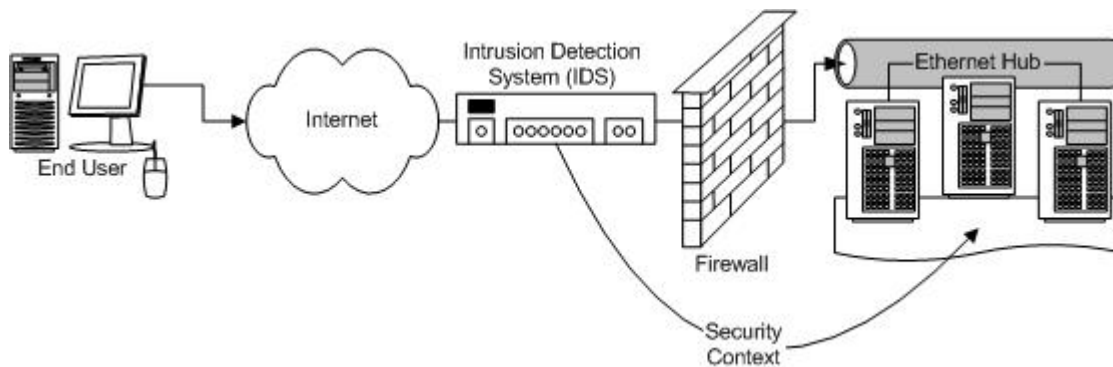


Figure 3-5: Server Farm Protection with an Intrusion Detection System

There are circumstances that justify selectively ignoring requests from certain IP addresses automatically, and your application code may be unable to gather evidence of the occurrence of these circumstances if your server farm is simply firewalled. However, deploying an IDS product that supports custom application notifications of intrusion attempts and developing a custom solution to integrate that counter intelligence into Web application request processing throughout your server farm is presently a complex, time-consuming task without industry standards. Further, creating rules for triggering automatic lockdown of IIS when an IDS does detect an attack in progress can be very tricky business. For one thing you don't want to give an intruder who gains control of one node in your farm the ability to reconfigure intrusion detection parameters for the rest of the farm.

Attacks aren't difficult to detect in well-designed Web applications. You can assume that a request for a URI that doesn't exist on the server is an attack, just as you can assume that a request that provides any name/value pair URL-encoded parameter values or FORM POST body elements that violate input validation rules or fall outside the list of known, valid FORM field names is an attack, provided that the Web application developer carefully tested every link and completed a thorough quality assurance test and took the time to document the FORM fields actually used in the application. A non-malicious user sending HTTP requests to IIS using a regular Web browser client program will be incapable of sending, accidentally, either a request that results in a 404 error or a request that violates an application's input validation rules without manually tampering with the URL or FORM POST request body. This assumes, of course, that the Web developer or content author is careful never to leave dead links either from page to page within a site or, by expiring URI's and failing to configure an automatic redirect, to allow search engines and links on other sites to reference pages that no longer exist. A user who manually tampers with their browser URL is obviously looking for security holes, as is any request that tries to POST FORM fields that don't exist and have never existed in the HTML FORMs served by a site. With a little preparation in advance, you can configure IIS to block further requests from any IP address that sends a request that fails input validation rules. Malicious traffic is also easy for an IDS to detect because any request or packet that matches known malicious requests or packets is obviously a malicious attack, and the IDS contains a database of known malicious attacks against which to compare.

Client Proxy Farms

The problem is that it might not be appropriate to immediately block further traffic from an IP address just because it sends a malicious attack without knowing more about the IP address. If the IP address belongs to a large proxy farm, essentially the opposite of a server farm where load balancing is performed on the client side, like the ones operated by Microsoft and America Online then you will block non-malicious requests from innocent users in order to block the potentially-harmless yet obviously-malicious requests from the attacker. Worse yet, you'll block all potential future traffic from all potential future users of that proxy farm long after the malicious attacker has moved to a new IP address. More context is often needed for a lockdown decision than the simple "attack detected; block IP address" logic that can be coded easily into automated countermeasures. For example, you might determine whether the IP address in question belongs to a dialup address block, is part of a known proxy farm, is likely a proxy server based on observation of multiple authentic users visiting your farm from the IP address simultaneously in the past, or is assigned to an organization that doesn't function as an ISP on behalf of other people, in order to decide that it is acceptable to temporarily block an IP address.

The nature of the attack from the IP address is also context for making a decision about the appropriate countermeasure. A DoS attack that can be stopped instantly by blocking the IP address at the firewall or IDS may impact other users who depend on the proxy server located at that IP address, but it's clearly better to DoS a few users than to DoS everyone. Since a well-managed and properly-security-hardened proxy server will detect and halt malicious activity itself rather than relay it to your server in the first place, you may decide that blocking IP addresses that send malicious traffic to your server farm is a completely acceptable response regardless of who or what the IP address represents. If large proxy farms like those operated by Microsoft and America Online for the benefit of their users block well-known malicious attacks then non-malicious users who depend on those companies' proxy farms don't have to worry about being denied access to your server farm by its policy of automatically blocking IP addresses from which attacks originate.

Dynamic Traffic Filtering by IP Address

Your server farm will most likely block packets sent to any port other than 80 (HTTP) or 443 (SSL over HTTP) at the firewall or IDS. This leaves your IIS boxes exposed only to HTTP requests from any IP address that can route packets to your farm from the WAN and any network traffic that comes in through the LAN. Unless your firewall or IDS performs application protocol validation on the packets it lets pass from the WAN, packets addressed to one of these open ports that carry malicious payloads other than requests that comply with the HTTP protocol are also a concern. IIS must attempt to make sense out of the contents of such packets, which will at least consume system resources. Application protocol validation is relatively easy for a firewall or IDS to conduct on HTTP communications compared to SSL over HTTP because with the exception of the SSL handshaking that occurs in the clear, everything else is encrypted within each TCP packet of an SSL-encrypted HTTP connection. IIS must attempt to decrypt data it receives from clients that are allegedly engaged in SSL-secured HTTP connections, and there is a possibility of attack by a specially-designed program that establishes an SSL connection and then throws bytes at IIS that have a malicious impact.

TCP packets that contain SSL-encrypted payloads can't be filtered out by the firewall or IDS based on automated application protocol validation because only the IIS box that performed handshaking and key exchange with the client has the ability to decrypt these packets. It is therefore necessary to deploy an additional IDS layer on each IIS box that gets involved in and has a chance to veto every request based on IDS rules, lists of blocked IP addresses, and any other security context information available. Ideally this IDS layer would also feed information back into the farm's firewall or IDS so that other NLB nodes in the farm aren't bothered with traffic from the same malicious source. A peer to peer (P2P) architecture could be deployed to enable each IDS module built into IIS on each NLB node to share intrusion information with other nodes. However, any communication that originates from a server farm node should be trusted minimally with respect to automating actions on other nodes or in remote locations because the node could be hijacked and the information could be forged. Sending intrusion details to other nodes or sending information to a master IDS for the entire farm about intrusion attempts or incidents in an automated manner is a good idea, but a human security administrator needs to manually authenticate any instructions that the entire farm will follow. A trained human brain will pick out suspicious patterns in moments with much greater accuracy than code a programmer spends months writing, and there's no trustworthy way for a node to conduct its own automated forensic audit and investigate an alleged incident in order to determine its authenticity before acting upon information supplied by another NLB node.

Ideally, an investigation to confirm the intrusion attempt will be performed by a human prior to blocking an IP address throughout the farm.

However, exposing a server farm shutoff switch to an intruder who succeeds in hijacking a single node in the farm may be an acceptable risk considering the additional security that is achieved through automatic IP address filtering for the entire farm as an intrusion countermeasure. Whether you permit peer notification of IP addresses that need to be blocked or implement filtering only as an administrative operation triggered securely from a central location by the administrator after an attack has been confirmed, you need to deploy an IDS layer inside IIS on each NLB node that listens on the internal network for instructions to block requests from a particular IP address. The IDS layer should not listen for these instructions on the network that routes traffic to and from the Internet using the NLB primary IP address, it should bind only to the interface that carries traffic such as heartbeat messages between the NLB nodes using the node's dedicated IP address. This is the preferred network architecture for Windows NLB, though you can use only one network if you choose; beware that traffic that would interfere with the NLB heartbeat could aggravate a DoS condition and prevent NLB convergence in the cluster.

The following C# code illustrates one possible implementation of source IP address filtering as a protective layer to allow IIS to block requests from malicious sources. The ability to respond with a countermeasure such as dynamic IP address filtering when an attack condition is detected is the first and most important part of an IDS layer for IIS. Your firewall or network IDS may have such a filtering feature also, but unless you have a secure way to automate updates to the configuration of the IDS for the entire farm from a node that first detects an attack that the IDS let through, the additional protective ability of an IIS IDS layer is an important defense countermeasure. Additional features of such an IDS layer would be to report intrusion details to an administrator automatically and the ability to receive additional administrative instructions such as authorization to remove a

particular IP address from the filter list. The code shown here is meant only to get you started as a basis for creating your own full-featured IDS layer that implements the security policy and automated countermeasures most applicable to your farm.

To halt IIS request processing at a lower level for applications and content not served by ASP.NET you can implement an ISAPI IDS instead as shown in Chapter 11, or purchase an IDS that is implemented as a device driver layer where monitoring network traffic is more efficient and more comprehensive. A simplistic yet effective IDS is better than none, and the additional processing overhead is a necessary and reasonable price to pay for improved security. Chapter 5 shows you how to configure the ASP.NET script engine to process all content types served by IIS, including static HTML files that contain no server side script. To deploy the simplistic ASP.NET-based IDS shown here so that it protects all files served by IIS requires following the instructions found in Chapter 5. The code shown is useful whether your IIS deployment is an isolated server island or a NLB server farm.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Web;
using System.Collections;
namespace IDSIIIS {
public class IDSIIISModule : System.Web.IHttpModule {
private TcpListener listen;
SortedList IPAddresses = new SortedList(64);
public void Init(HttpApplication context) {
context.AuthorizeRequest += new EventHandler(this.AddressFilter);
ThreadStart startlistening = new ThreadStart(NetListener);
Thread listeningThread = new Thread(startlistening);
listeningThread.Start(); }
public void Dispose() {}
public void AddressFilter(object sender,EventArgs e) {
bool bSearch = true;
HttpApplication app = (HttpApplication)sender;
String addr = app.Request.UserHostAddress;
while(bSearch) {
try {
if(IPAddresses.Contains(addr)) {
throw(new Exception("BLOCKED")); }
bSearch = false; }
catch(Exception ex) {
if(ex.Message == "BLOCKED") {
bSearch = false;
app.CompleteRequest(); }}}
private void NetListener() {
byte[] buf;
int bytes;
bool loop = true;
```

```

TcpClient tcp;
NetworkStream net;
IPAddress dedicatedIP = IPAddress.Parse("192.168.0.1");
listen = new TcpListener(dedicatedIP,8080);
listen.Start();
while(loop) {
try {
tcp = listen.AcceptTcpClient();
net = tcp.GetStream();
buf = new byte[tcp.ReceiveBufferSize];
bytes = net.Read(buf, 0, buf.Length);
tcp.Close();
try {
String sIP = System.Text.ASCIIEncoding.ASCII.GetString(buf);
IPAddress.Parse(sIP);
IPAddresses.Add(sIP,DateTime.Now); }
catch(Exception badaddress) {} }
catch (Exception readerror) {
loop = false; }}}}

```

In the C# ASP.NET IDS code shown, dedicatedIP should be set to an appropriate value on each NLB node. The address 192.168.0.1 is shown hard-coded only as an example place-holder. Hard-coding the dedicated IP address works fine if you are willing to perform a different build for and deploy different code to each node, otherwise you'll need to write a simple algorithm to dynamically discover the dedicated IP address you've assigned to the node in a manner that is appropriate for your farm's particular dedicated addressing scheme. For instructions on deploying a System.Web.IHttpModule interface module in ASP.NET see Chapter 5. The network listener thread created by the IDS IHttpModule is designed to receive from a TCP data stream a single IP address that is to be added to the list of blocked IP addresses. The following C# code can be used as an administrative command line utility or adapted for inclusion within any .NET application to send additions to the blocked IP address list maintained by the IIS IDS layer.

```

using System;
using System.Net;
using System.Net.Sockets;
using System.IO;
namespace IDSIIISModuleManager {
class IDSManger {
[STAThread]
static void Main(string[] args) {
String node, blocked;
if(args.Length > 1) {
node = args[0];
blocked = args[1];
TcpClient tcp = new TcpClient(node,8080);
StreamWriter writer = new StreamWriter(tcp.GetStream());
writer.Write(blocked);
writer.Close();
}
}
}

```

```
tcp.Close(); }}}
```

It doesn't take much additional code to add encryption to the instruction to add addresses to the blocked address list in order to add authentication that rejects unauthorized changes. See the Cryptology and Intrusion Countermeasures section later in this chapter for more on this topic. Once you deploy custom code or a third party product that gives you the ability to automate dynamic filtering of request traffic based on IP address or other request identifying characteristic, the real work begins. Determining what conditions will trigger automated filtering so that manual administrator intervention isn't the only incident response you have planned when your server farm comes under attack and coding automated recognition of those conditions are the final steps. The next two sections explain these steps in more detail.

Detecting Scans, Probes, and Brute Force Attacks

Trespassing in your server farm through scans and probes or repeated authentication failures that exceed a reasonable request rate and may indicate a brute force attack on password security can easily be detected by an IDS. Requests for application services from such IP addresses must be blocked automatically or at least given little trust, just as in the physical world a security guard would reject the credentials of a 12-year-old dressed in a business suit who claims to be the company CEO after the security guard observes the 12-year-old climbing fences and trying to open locked doors. Obviously the real CEO, even if he or she does happen to be 12 years old, would not exhibit such suspicious trespassing behavior before requesting access through the front door based on authentic identification credentials. Further, the CEO would never show a security guard a million fake ID cards that fail the security guard's authentication criteria until stumbling randomly upon an ID card that satisfies the security guard's criteria. The security guard would know after examining the first fake ID that the person is an imposter.

When excessive authentication failures occur over a very short period of time those failures are always evidence of malicious activity. This is especially apparent without further confirming evidence when the authentication failures pertain to a single user ID and the time between failures is so small that only an automated system could be sending the authentication credentials so rapidly. Automatically blocking further requests from any IP address that appears to be sending requests to your server farm faster than a human user would reasonably be capable of doing so using regular Web browser software is a valid automated security policy IDS rule. These rules are sometimes referred to as rate filters, and implementing such rate filtering is often an application-specific responsibility due to the differences in ways to facilitate authentication from one application to another. Some applications will consider rapid-fire Web client robots to be permissible, as long as the robots behave and follow rules outlined in robots.txt and META tags. When triggered, application context sensitive rate filters automatically block further request processing by IIS from the offending source IP address using dynamic traffic filtering IDS code like that shown in the last section. In addition to rate filters based on IP address, which can be problematic if many users sharing the same proxy server are locked out because of the malicious activity of others, session lockouts and user lockouts or temporary permissions reductions are appropriate application-specific security countermeasures. Innocent users caught by a rate filter understand a temporary lockout or reduction in privileges implemented for their own protection.

Deploying Honeypots to Detect Intruders

A honeypot is a network node that exists only to attract intruders. Any access attempt to the network node must be malicious because there is no legitimate reason for any person or program to access the node. Some honeypots are set up with intentional vulnerabilities so that attackers are able to penetrate them and install malicious code that can then be analyzed to determine whether it represents a new type of attack or something that has been observed previously. This is one of the techniques employed to keep tabs on the real threat level that the network poses at any given time. Scans often iterate through address ranges in a predictable way, and placing honeypot nodes immediately above and immediately below, as well as in between, the address range used by your farm can catch scans before they reach your production addresses and enable your IDS to shield your farm from malicious requests that may follow scans. The following code illustrates a honeypot program that adds addresses that scan TCP ports 1 to 150 to the list of blocked addresses in the IIS IDS layer shown previously.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Web;
using System.Collections;
using System.IO;
[STAThread]
static void Main(string[] args) {
    IPAddress ip = IPAddress.Parse("127.0.0.1");
    int a;
    Socket s;
    IPAddress remote;
    TcpListener[] tcp = new TcpListener[150];
    IDSIISModule ids = new IDSIISModule();
    ThreadStart startlistening = new ThreadStart(ids.NetListener);
    Thread listeningThread = new Thread(startlistening);
    listeningThread.Start();
    for(a = 1;a < tcp.Length;a++) {
        try {
            tcp[a-1] = new TcpListener(ip,a);
            tcp[a-1].Start(); }
        catch(Exception e) {} }
    while(true) {
        for(a = 1;a < tcp.Length;a++) {
            if(tcp[a-1].Pending()) {
                s = tcp[a-1].AcceptSocket();
                try {
                    remote = IPAddress.Parse(((IPEndPoint)s.RemoteEndPoint).Address.ToString());
                    TcpClient t = new TcpClient("192.168.0.1",8080);
                    StreamWriter writer = new StreamWriter(t.GetStream());
                    writer.Write(remote.ToString());
```

```
writer.Close();  
t.Close();  
s.Close();  
ids.ShowAddresses(); }  
catch(Exception neterr){s.Close();}}}}}
```

The honeypot program binds to 0.0.0.0, the address that automatically maps to all network adapters in the system, with the idea that the program is running on a dedicated honeypot node. However, the honeypot works well as a process running alongside IIS on a regular NLB node provided that the node is configured with multiple IP addresses including both the dedicated NLB address and the honeypot address. In this case you would bind the honeypot to the honeypot address only instead of to 0.0.0.0 which allows the honeypot to receive connection requests addressed to any IP address bound to any adapter in the system. To deploy this code as a real honeypot so that it sends IP addresses to all NLB nodes in the farm, you can hard-code the dedicated IP addresses of each node into a loop that establishes a connection with each node one at a time using TcpClient until all nodes' IIS IDS layers have been notified of the remote IP address to block. The honeypot code can also be added as another thread to the IIS IDS layer, so that the IDS functions as a honeypot also.

A honeypot will only be able to receive traffic directed to ports that are open in your farm's firewall, so if port 80 is the only port allowed then that's the only port to which honeypots must listen. Other ports are optional, but can be valuable as a way to detect security failures in your firewall if it ever improperly allows packets to reach the LAN from the WAN with a port number that is supposedly filtered.

The security policy "if you scan me, I will refuse to service requests from you" may seem somewhat drastic, especially when you consider the fact that non-malicious users will most likely be assigned the IP address at some point in the future if the address is part of a DHCP address block or may already be using the IP address through a shared client proxy farm, but it provides a good starting point for intrusion detection and automated response. Fine tuning the automated response, or sending honeypot captures to an administrator for further review and allowing the administrator to trigger address blocking only after manual confirmation that it is justified, are all possibilities once you get your IDS deployed throughout the server farm. One idea worthy of mention is to send a remote address that has been blocked an HTTP response that provides the user with an apologetic note explaining that the server has detected malicious network traffic from the address they are currently using and offer contact information and a procedure for the user to follow to request that the IP address be removed from the blocked list. When an attacker sees this message immediately after violating your IDS security policy rules they may be inclined to move on to an easier target rather than poke and prod your server farm from hundreds or thousands of IP addresses in order to search for vulnerabilities and create a DoS condition for anyone else who may share the IP addresses used by the attacker. Giving attackers such an easy way to create small-scale DoS conditions may seem odd at first, but after giving it more consideration you may conclude that giving attackers the ability to produce small temporary DoS outages that also remove them as a threat is far better than giving attackers opportunity to take control of the farm.

Cryptology is the study of cryptography, which literally means “secret writing”, and cryptanalysis, which is the theory and technique of analyzing cryptography to discover vulnerabilities, decrypt ciphertext by discovering the decryption key or deducing its contents without full decryption, or quantify information security through analytical proof. Many automated intrusion detection mechanisms rely on principles of cryptology where most manual intrusion detection is based more on common sense and the gathering of electronic evidence that a security administrator knows should not appear on a system that has not been compromised. The procedure followed by an administrator to gather such evidence can be automated with some confidence, and pattern recognition algorithms facilitate interpretation of suspicious or malicious log file, network traffic, memory usage, and program code patterns by security analysis software. However, assembling a database of suspicious patterns and optimizing algorithms that scan for those patterns automatically in code and data is an infosec specialty best left to antivirus and intrusion detection software vendors who can devote large budgets and staff to the task. You still have to use common sense to pick out everything that isn’t recognized automatically and actually read log files even when you have the best automated systems scanning for threats continuously. There are several things you can and should do yourself to apply well-known principles of cryptology and specific tools of cryptography to increase server farm security and automate intrusion detection in application-specific and deployment-specific ways.

Digital Signatures and Credential Trust

Asymmetric key cryptography, where a key is used for encryption that is useless for decryption, enables one-way transformations that form the basis of digital signatures. The proof of trust that a digital signature offers is directly dependent on the difficulty an attacker would have in discovering the key that was used in the original transformation assuming the key is successfully kept secret by its owner. However, key theft is not the only vulnerability that impacts digital signatures because digital signatures are not unforgeable. Given enough time and cryptanalytical resources, the secret can be discovered based on a brute force key search. The simplest way to think about brute force cryptanalysis is to imagine millions of powerful computers operating full-time in parallel to scan unique portions of key space by repeatedly applying the digital signature algorithm until a matching signature is found. The length of time it would take to discover the right key using all computing power in existence today if it was all working exclusively and in parallel on this one problem is said to be so large as to represent proof that the probability of anyone succeeding in this cryptanalysis in their own lifetime is infinitesimal. This means there is a chance, however small, of successful cryptanalysis that would give an attacker the ability to forge digital signatures.

Whenever you build or deploy a system that places blind trust in the validity of digital signatures you increase the chances, however slightly, that an attacker will be able to penetrate your system’s security. When the best security your system has to begin with is a relatively short administrator password, it may seem ridiculous to worry about the threat of forged digital signatures. It’s important to be aware, however, that the practical nature of the threat is so different as to be difficult to compare with password security. Digital signature secret key cryptanalysis occurs in secret, and only the cryptanalyst knows of the effort to break a code or of its success. Brute force password cracking, on the other

hand, produces observable signs of attack. Given a choice between weak password protection that you will always observe being attacked and a nearly-impenetrable protection that you will never observe being attacked, which would you select? The decision to grant unlimited trust to automated systems based on automated validation of digital signatures starts to look quite a bit more risky when you consider it from this perspective compared to granting unlimited trust to a user of an automated system based on verification of the user's credentials that can only be brute forced by an attack that is certain to be observed. As soon as such an attack is detected, and on a regular basis regardless, old credentials are invalidated and new credentials issued, rendering useless any progress an attacker might have made in brute forcing passwords and forcing them to start over again from scratch. Best-case for the attacker this means capturing another encrypted copy of a password database or a credential transmitted over a secure channel to a password-protected resource and restarting cryptanalysis. With strong passwords and strong encryption, periodic password changes offer excellent practical security provided that your systems are configured to detect any attempt to brute force passwords or intercept encrypted passwords for cracking through cryptanalysis.

To begin brute forcing a digital signature key, an attacker need only to intercept a sample of an authentic digital signature and the message to which it was applied. Unlike encryption, which hides the content of the message, digital signatures are often used in such a way as to reveal the entire message to anyone who receives it, with the idea that the most important thing is to enable the recipient to verify the digital signature applied to the message not protect secrecy of the message. In all such cases, a cryptanalyst who receives a copy of the digitally signed message has everything required to begin a brute force attack attempting to discover the signature secret key. This substantially increases the likelihood that anyone will ever even try such improbable cryptanalysis compared to a problem that is substantially more difficult, such as discovering a secret key used to produce a digital signature that an attacker can't even read until they first succeed in decrypting the ciphertext inside which a digital signature is enclosed along with a secret message. This type of combined encryption and digital signature, which is easy to accomplish if both the sender and receiver have asymmetric key pairs and have exchanged public keys prior to secure communication, is significantly more secure than digital signatures that are meant for anyone and everyone to verify as attachments to messages sent in the clear. This reality foreshadows a computer security disaster of epic proportions as software vendors optimistically deploy automatic update systems that perform only digital signature verification without encryption and rely on the same key pair for digitally signing communications with large numbers of receivers in a one-to-many manner.

Ideally key pairs used in digital signatures are expired frequently and replaced with new keys. How frequently depends on how sensitive the data and how paranoid the user. It also depends on what the key pair has been used for since it was generated and issued. The interesting thing about digital signatures is that they are only one possible use of asymmetric cryptography and asymmetric key pairs. A digital signature by itself doesn't prove identity, any more than carrying a piece of paper with somebody else's signature on it into a bank proves you are the person whose printed signature you offer to the bank teller. Digital signatures can establish a likelihood that a particular person authored or signed a particular document at some point in the past, but that's only useful for certain applications, and only so long as the signature secret key remains secret.

Proving that you have the ability to encrypt a dynamically-selected message using a first key that corresponds to a known second key so that the recipient of the encrypted message can decrypt and validate the message using the second key is where the proof of identity occurs. Assuming, of course, that the recipient is sure they are in possession of an authentic copy of the matching key from the key pair. This real-time use of digital signatures for identity authentication corresponds to affixing a written signature to a document in the presence of another person in order to satisfy that person that you are capable of producing, on demand, writing that corresponds visually to the signature associated with a particular identity. But this is just a real-time short-lived variation of a digital signature; asymmetric cryptography offers another valuable tool that is especially useful in managing server farm security: long-lived one-way encryption that does not include hashing or the production of a message authentication code (MAC) for the purpose of applying a digital signature to particular data but instead encrypts data in bulk without the use of a symmetric key algorithm. Bulk encryption using an asymmetric cipher is slower than bulk encryption using a symmetric cipher, but the inability for the encrypting computer to decrypt the resulting ciphertext makes asymmetric ciphers an important tool for bulk encryption in server farms.

Asymmetric Cryptography for Bulk Encryption

Data encryption using a programmable computer and symmetric encryption is confronted by a bewildering catch-22. If the programmable computer isn't proven to be 100% secure, as virtually no programmable computer in use today is at the time of its use due to the fact that using a programmable computer can lead directly to changes in the computer's programming, then how do you know that the encryption was performed correctly and the encryption key kept secret while in memory? If the computer turns out to be compromised and its secret encryption key was intercepted, then all ciphertext produced using that key is accessible to the attacker who intercepted the key and the act of encrypting data only reduces the number of malicious third parties who can read the data when they intercept its new ciphertext transformation. This means your programmable computer has to be trustworthy whenever you encrypt data. If it isn't trustworthy, future encryption is pointless and all existing ciphertext transformations that resulted from previous use of the secret keys accessible to the untrustworthy computer are also compromised. If the computer is trustworthy now and it will continue to be in the future then what's the point of encryption? If you plan for security incidents with the idea that every key accessible to your server farm may be compromised by an intrusion, it doesn't take long to realize that your long-lived sensitive data such as customer financial and personal information that you have an obligation to protect but also have a long-term need to access is inadequately protected through application of a symmetric cipher because the resulting ciphertext can be decrypted using the same key that was used to encrypt, and the compromised server farm contains a copy of that secret symmetric key. A worst-case security incident scenario where the entire server farm is physically stolen leads to a likelihood of malicious decryption of this sensitive data that isn't acceptable. You can store the data off-site and carefully avoid writing any persistent plaintext or ciphertext data to the farm's hard drives, but that only forces determined thieves to leave your server farm alone and steal different computers instead.

Whenever it's necessary for a server farm to automatically encrypt data but unnecessary for it to ever automatically decrypt that data, symmetric encryption is the wrong tool for the job. Symmetric encryption functions like a simple authentication credential; possession of the secret key is equivalent to authorization to access the plaintext data at will. Without that authentication credential, access is denied. When a human user want to protect information for "eyes only", symmetric key ciphers are a good solution. The human user will keep a copy of the encryption key to use for future decryption and only give a copy to others who are authorized to access the information. But a computer that has an automated ability to retrieve and use a symmetric key has less control over who might end up with a copy of that key than does a human user. The computer code that performs encryption or decryption can be forced to do the bidding of a malicious third-party who controls its execution. Simple well-known tricks resembling buffer overflow attacks where return addresses, pointers, or subroutine entry-points are overwritten with malicious values can enslave the cryptographic functionality of the authentic code and put it to work for a third party or reveal the symmetric key used by the code. This is less of a concern when a symmetric cipher is used on-demand using an encryption key supplied by the user, but malicious code could be present during that one-time event, too, that would compromise the key.

Asymmetric ciphers offer superior protection because it's irrelevant to the security of the ciphertext whether or not a third party intercepts the key used in the cryptographic transformation. Compare this to the typical applications of asymmetric ciphers, digital signatures where data security isn't the point but rather the point is to allow anyone to decrypt the ciphertext in order to verify the digital signature, or the use of an asymmetric cipher merely to encrypt a symmetric key that was used for bulk encryption. The idea in the latter application is to facilitate secure symmetric key exchange with the recipient of the ciphertext produced using a symmetric cipher. The reason the asymmetric cipher isn't used in the first place for bulk encryption of the data that the recipient receives is that the symmetric cipher transformation takes substantially less computing power and therefore time. This is a poor excuse for treating sensitive data with less than optimal protective care and rigor.

Creating Public Key Cryptostreams

Examine the ways in which malicious code, if it were running on and in complete control of a compromised system, would steal sensitive data that passes through the system unencrypted. One way would be for the code to copy plaintext before it gets encrypted on the way out or on the way into secure storage. The copy could be sent over the network to the attacker or saved to a file for access later, either of which creates the potential for the malicious activity to be detected even if the presence of the malicious code is not because the unauthorized data transmission or unexplained file would call attention to the theft in progress. Another option that is simpler to conceal even though it may be harder to code is for the secret key to be intercepted at runtime so that ciphertext can be decrypted using the key at a time in the future. For an attacker who can obtain ciphertext relatively easily, such as an employee or other insider, key theft is the higher priority anyway. Given the choice of stealing a 128-bit symmetric encryption key out from under the nose of a watchful security administrator or stealing 128 megabytes of plaintext data, an attacker who can get at ciphertext another way or at another time would choose to steal the small amount of data and worry about getting the ciphertext later. Asymmetric

encryption used to transform 128 megabytes of plaintext leaves theft of the original plaintext data as the only attack option, making it easier to implement theft detection countermeasures in network devices like an IDS or reverse proxy that isn't compromised. To deploy an asymmetric key encryption solution for a server farm requires the secure generation of many key pairs. Microsoft .NET makes this simple with the class `System.Security.Cryptography.RSACryptoServiceProvider` and two lines of code.

```
new RSACryptoServiceProvider(1024).ToXmlString(true);
```

The parameter passed to `RSACryptoServiceProvider`'s constructor is the key size in bits, and the parameter passed to method call `ToXmlString` is a Boolean indicating whether or not the XML string should include the private key. The resulting XML represents a valid key pair. It can be imported at runtime into an instance of `RSACryptoServiceProvider` using the `FromXmlString` method. The following code shows how to integrate the notion of one-way asymmetric encryption using public key cryptostreams into the console application shown previously that was designed as an administrative tool to dispatch IP addresses to IIS IDS layers in the NLB nodes of the server farm. This example is meant to complete the illustration of a straight-forward and secure method for sending IDS configuration instructions from an administrative node that has knowledge of the public key assigned to each NLB node in the server farm. It isn't the best example of the protection of critically sensitive data flowing through a NLB node, since this example shows data flowing to the NLB node not through it, but you should be able to see how to adapt this code to that purpose easily based on this example. Another reason to show this example rather than start a new one from scratch is to reinforce the versatility of asymmetric cryptography; by generating a key pair in advance and deploying the public key only to a single sender location, a receiver in possession of the private key can authenticate the source of the data as coming from the sender to whom the public key was issued.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Security.Cryptography;
using System.IO;
namespace IDSModuleManager {
class IDManager {
[STAThread]
static void Main(string[] args) {
FileStream f;
NetworkStream o;
int bytes;
byte[] buf;
CryptoStream csTransform;
RSACryptoServiceProvider rsaEncrypt = new
RSACryptoServiceProvider(1024);
rsaEncrypt.FromXmlString("<RSAKeyValue><Modulus>" +
"vuQkEFfmNf/XTIRL/ga4WYBsA2GMqIpUpwPmCEBWIQGwXfRioppWTdIWz0" +
"1u6o4h8R38aInfbh7erO/O+anmgbfHdCf+8oc5G0WcCU1AYp7hV5rBHQ4g" +
"b0oalHi+RCKkcrvzQ2PZjchLcDfN15SOgsXDf88fdxFzUoZA23RXrbs=" +
"</Modulus><Exponent>AQAB</Exponent></RSAKeyValue>");
```

```

RSACryptoStreamTransform rsaTransform = new
    RSACryptoStreamTransform(rsaEncrypt);
String node;
if(args.Length > 1) {
node = args[0];
if(File.Exists(args[1])) {
try { TcpClient tcp = new TcpClient(node,8080);
o = tcp.GetStream();
f = File.Open(args[1],FileMode.Open);
buf = new byte[rsaTransform.OutputBlockSize];
csTransform = new CryptoStream(
    f,rsaTransform,CryptoStreamMode.Read);
DateTime start = DateTime.Now;
while((bytes =
    csTransform.Read(buf,0,buf.Length)) > 0)
    {o.Write(buf,0,bytes);}
DateTime finish = DateTime.Now;
csTransform.Close();
f.Close();
o.Close();
tcp.Close(); }
catch(Exception ex)
{System.Console.WriteLine(ex); }}}
class RSACryptoStreamTransform : ICryptoTransform,IDisposable{
public bool CanReuseTransform {get {return(true);}}
public bool CanTransformMultipleBlocks {get {return(false);}}
public int InputBlockSize {get {return(117);}}
public int OutputBlockSize {get {return(128);}}
private RSACryptoServiceProvider rsaEncrypt;
public RSACryptoStreamTransform() {}
public RSACryptoStreamTransform(RSACryptoServiceProvider rsaCSP)
{rsaEncrypt = rsaCSP;}
public void Dispose() {}
public int TransformBlock(
byte[] inputBuffer,int inputOffset,int inputCount,
byte[] outputBuffer,int outputOffset) {
byte[] plaintext = new byte[inputCount];
Array.Copy(inputBuffer,inputOffset,plaintext,0,inputCount);
byte[] ciphertext;
ciphertext = rsaEncrypt.Encrypt(plaintext,false);
ciphertext.CopyTo(outputBuffer,outputOffset);
return(ciphertext.Length); }
public byte[] TransformFinalBlock(
byte[] inputBuffer, int inputOffset, int inputCount) {
byte[] plaintext = new byte[inputCount];
Array.Copy(inputBuffer,inputOffset,plaintext,0,inputCount);
byte[] ciphertext;
ciphertext = rsaEncrypt.Encrypt(plaintext,false);
return(ciphertext); }}}

```

This code is similar to that shown previously in this chapter. It creates a console program that connects to the IIS IDS layer's listening socket on port 8080 in order to tell the IDS to block a list of IP addresses. This version of the program reads a text file, encrypts its contents using the public key assigned to the IDS, and sends the resulting ciphertext over the network connection. To enable the application of an asymmetric cipher to the task of performing bulk encryption, a non-traditional use, the code creates a helper class called `RSACryptoStreamTransform` that implements `ICryptoTransform` and can therefore be passed to the `CryptoStream` constructor. The `CryptoStream` object and its encapsulated instance of `RSACryptoStreamTransform` take care of encrypting bytes read from the file input stream using 117 byte plaintext blocks. Each block is transformed using the RSA asymmetric cipher.

By encrypting each 117 byte block of data read from the file input stream the code emulates a symmetric block cipher but applies an asymmetric transformation instead of symmetric key encryption to each plaintext block. A symmetric block cipher would transform plaintext faster, but the RSA encryption algorithm isn't so slow that it won't work well in most applications. Decryption of RSA ciphertext, however, takes many times longer than encryption so there are applications that won't work well using RSA as a real-time asymmetric block cipher unless the receiver has a lot more computing power and speed available than the sender. Whenever a block cipher transforms data one block at a time using the same encryption key over and over again patterns can emerge in the resulting ciphertext output because each identical block of plaintext input will transform to the same ciphertext. To prevent such patterns, which make cryptanalysis much easier, symmetric block ciphers commonly introduce feedback that scrambles each block in an unpredictable way before encrypting it. The feedback has to be unpredictable to a cryptanalyst but not unpredictable to the cipher. To make this sort of feedback possible, an initialization vector (IV) is used to scramble the first block of plaintext. The IV becomes a second secret that must be protected and conveyed securely to the recipient along with the symmetric encryption key. Each block transformation introduces feedback into the next block, preventing patterns in the final ciphertext. When no feedback is introduced and no IV is used, the block cipher is said to be operating in Electronic Code Book (ECB) mode. Each of the symmetric ciphers provided by Microsoft .NET operate by default in Cipher Block Chaining (CBC) mode, where an IV is used in an exclusive OR transformation of the first plaintext block to seed the feedback.

`CryptoStream` in Microsoft .NET assembles plaintext blocks and calls into an `ICryptoTransform` object while passing the current block that needs to be transformed. The `InputBlockSize` and `OutputBlockSize` property accessors in the `ICryptoTransform` interface allow `CryptoStream` to determine the proper size for input and output buffers sent to and received from any `ICryptoTransform` object. The RSA algorithm adds a minimum of 11 bytes of padding in each transformation, so an output block size of 128 bytes will be produced from only the first 117 bytes of a plaintext buffer. For maximum efficiency and for the RSA `CryptoStream` technique to actually work, the `RSACryptoStreamTransform` class shown hard-codes the input and output buffer sizes accordingly to leave room for those 11 bytes per transformation. Ciphertext output block sizes, in bits, produced by block ciphers are always equal in size to the number of bits in the encryption key. Any `OutputBlockSize` can be used in your implementation that is a multiple of the key size but for RSA the `InputBlockSize` must be equal to or smaller than

the `OutputBlockSize` minus 11 bytes for padding. A small input block size compared to output block size results in more padding by RSA.

As long as no other sender is given a copy of the public key from that particular key pair, it functions exactly like an authentication credential and proper digital signature while providing strong encryption. The receiver can't prove that the sender was the only possible sender, as with a digital signature, because the receiver also knows the public key as does the administrator who configured this manual trust relationship, but that doesn't matter because the goal is encryption not signatures. The example shows how to authenticate the validity of an instruction sent between automated systems and demonstrates an interesting lightweight digital signature in the context of a preconfigured administrative trust function for a server farm. To enable other senders to send instructions to the receiver that will authenticate in the same way, the public key can be given to additional senders. In this way many-to-one trust relationships can be formed based on a single asymmetric key pair where any sender who has the public key can issue commands the receiver will accept as authentic.

To adapt the code shown to function as previously described in this chapter for the purpose of securing data that passes through a NLB server farm node on its way to somewhere else, the code is simply deployed to the NLB node and it acts as the sender instead of the receiver. As senders, the NLB nodes can each be assigned different public keys so that there is a one-to-one trust relationship between sender and receiver based on the receiver's knowledge of the public key that was given to the sender and the sender's possession of that single key for securely communicating with the receiver. All senders can share the same public key in order to deploy a many-to-one relationship between many senders and the one receiver. The central premise of this asymmetric bulk encryption technique is that it is better to generate keys in advance and deploy them cautiously with built-in controls on the potential uncontrollable spread of sensitive plaintext information than to generate encryption keys on the fly and undertake dynamic key management. The dynamic generation of keys on a compromised system is self-defeating because the keys themselves are subject to theft, whereas a key pair generated elsewhere on a secure key generation workstation creates the potential to deploy half of the key pair, the public key, to automated systems that need to secure information to the best of their ability even in the case of an intrusion incident where keys are compromised.

Decrypting Public Key Cryptostreams

Only the key that corresponds to the asymmetric encryption key can be used to decrypt ciphertext produced using the encryption key. This key, the other half of the key pair, is referred to as the private key if the public key was used for encryption as shown in the previous example. A digital signature uses the private key for encryption and the public key for decryption. The ciphertext can flow in either direction, as the crypto math involved doesn't care about public vs. private or digital signature vs. bulk encryption. A cryptographic algorithm just transforms bits. It's very important to realize that just because a key is called a public key that doesn't mean you have to give it out publicly. The number of potential trust configurations are as large as the complexity of the system to which you apply asymmetric cryptography as an application feature.

In the examples shown in this chapter, the server farm IDS scenario is used because it offers a good real-world situation where a relatively small number of entities need to exchange data securely, and there are several distinct trust configurations. The most important configuration is the one in which each NLB node that receives sensitive information from users, credit card payment information for example, needs to prevent itself from accessing that information for any reason as soon as its need for that information ends. But the information has to be saved, and it should be encrypted. Asymmetric bulk encryption solves this problem very nicely. The IIS IDS layer that automatically blocks request processing from certain IP addresses could authenticate and encrypt administrative commands received over the network using symmetric key encryption instead, so it isn't the best example of an asymmetric trust model. The following code completes this example by showing how the IDS layer would decrypt, and thereby authenticate the contents of, an asymmetric ciphertext cryptostream received over the network.

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Web;
using System.Collections;
using System.IO;
using System.Security.Cryptography;
namespace IDSIIIS {
public class IDSIIISModule : System.Web.IHttpModule {
private TcpListener listen;
SortedList IPAddresses = new SortedList(64);
public void Init(HttpApplication context) {
context.AuthorizeRequest += new EventHandler(this.AddressFilter);
ThreadStart startlistening = new ThreadStart(NetListener);
Thread listeningThread = new Thread(startlistening);
listeningThread.Start(); }
public void Dispose() {}
public void AddressFilter(object sender,EventArgs e) {
bool bSearch = true;
HttpApplication app = (HttpApplication)sender;
String addr = app.Request.UserHostAddress;
while(bSearch) {try {
if(IPAddresses.Contains(addr)) {throw(new Exception("BLOCKED")); }
bSearch = false; }
catch(Exception ex) { if(ex.Message == "BLOCKED") {
bSearch = false;
app.CompleteRequest(); }}}}
private void NetListener() {
CryptoStream csTransform;
RSACryptoServiceProvider rsaDecrypt = new RSACryptoServiceProvider(1024);
rsaDecrypt.FromXmlString(
"<RSAKeyValue><Modulus>vuQkEFfmNf/XTIRL/ga4WYBsA2GMq" +
"lpUpwPmCEBWIQGwXfRioppWTdIWz01u6o4h8R38alnfbh7erO/O+anmgb" +
```

```
"fHdCf+8oc5G0WcCU1AYp7hV5rBHQ4gb0oalHi+RCKkcrvzQ2PZjchLcDf" +
"N15SOgsXDf88fdxFzUoZA23RXrbs=</Modulus><Exponent>AQAB</Exp" +
"onent><P>4LWluM82AHArYV3ojQ6Uzef3L5VBpn3y1wRvffg3j27w/KyB" +
"ou0Zo/LnqqBc885dfLqqaBEBewxLIEpoFfalhw==</P><Q>2XkPOpd" +
"Af6sbyML41pwNvZg2CXcc49DBYbamEW+I+xAFAvBSeMP6O09fqO0jN" +
"mdFeTAbACrQI7gfMteeP9JiLQ==</Q><DP>XV/yBWHNfdceytlkBiF2" +
"Ai4PEE3EbvwNOj4UmlLnu4mNSGHiqLI/wlnwnH1wwrsRLABhSUcvx1L" +
"voRpeMCo2xw==</DP><DQ>rhbSERYphMoGGjk2fp44BbFGeLdlGjqHw" +
"+AB+u0tW8XMLTkS3CgONdJpgolq8Q8kt0nCI5UinIHBP+MJhl+3FQ==" +
"</DQ><InverseQ>e9Bf8RurDeKstBP5Awmnc78WgBiaqVTVOpxx3YF" +
"fsG+Q3YHK1PgRkQKp8uMIHafAIQ0cEq7BxotXd5PYoTN2VQ==" +
"</InverseQ><D>iaZFgyt/K80y2VBE5AbAhHmgace8AATQCi" +
"c7hxOth9uJ7BY/0fTs6uzl2dKCeszHGPGAhMgN34CPHbFHVkz5M64" +
"QvimHE1imX3LPD7bWb00Kmd+G0CKJ6BUcreeYpQffcFT3FwO3fEFY" +
"g44j/2UGdU2RgMiUuvOT+DTO7Os+EtE=</D></RSAKeyValue>");
```

```
RSACryptoStreamDecipher rsaTransform = new
RSACryptoStreamDecipher(rsaDecrypt);
byte[] buf;
int bytes;
bool loop = true;
TcpClient tcp;
NetworkStream net;
IPAddress dedicatedIP = IPAddress.Parse("192.168.0.1");
listen = new TcpListener(dedicatedIP,8080);
listen.Start();
while(loop) {try { tcp = listen.AcceptTcpClient();
net = tcp.GetStream();
buf = new byte[rsaTransform.OutputBlockSize];
csTransform = new CryptoStream(
 net,rsaTransform,CryptoStreamMode.Read);
StringWriter plaintext = new StringWriter();
buf = new byte[tcp.ReceiveBufferSize];
while((bytes = csTransform.Read(buf,0,buf.Length)) > 0) {
plaintext.Write(
 System.Text.ASCIIEncoding.ASCII.GetString(buf,0,bytes));}
csTransform.Close();
net.Close();
tcp.Close();
try { String sIP = null;
StringReader reader = new StringReader(plaintext.ToString());
while((sIP = reader.ReadLine()) != null) {
IPAddresses.Add(IPAddress.Parse(sIP).ToString(),DateTime.Now);
}} catch(Exception badaddress){}}
catch (Exception readerror) {
loop = false; }}}
```

```
class RSACryptoStreamDecipher : ICryptoTransform, IDisposable
{public bool CanReuseTransform {get {return(true);}}
public bool CanTransformMultipleBlocks {get {return(false);}}
public int InputBlockSize {get {return(128);}}
```

```

public int OutputBlockSize {get {return(117);}}
private RSACryptoServiceProvider rsaDecrypt;
public RSACryptoStreamDecipher() {}
public RSACryptoStreamDecipher(
    RSACryptoServiceProvider rsaCSP){rsaDecrypt = rsaCSP;}
public void Dispose() {}
public int TransformBlock(
    byte[] inputBuffer, int inputOffset,
    int inputCount, byte[] outputBuffer, int outputOffset) {
    byte[] ciphertext = new byte[inputCount];
    Array.Copy(inputBuffer,inputOffset,ciphertext,0,inputCount);
    byte[] plaintext;
    plaintext = rsaDecrypt.Decrypt(ciphertext,false);
    plaintext.CopyTo(outputBuffer,outputOffset);
    return(plaintext.Length); }
public byte[] TransformFinalBlock(byte[] inputBuffer, int inputOffset, int inputCount) {
    byte[] ciphertext = new byte[inputCount];
    Array.Copy(inputBuffer,inputOffset,ciphertext,0,inputCount);
    byte[] plaintext;
    plaintext = rsaDecrypt.Decrypt(ciphertext,false);
    return(plaintext); }}}

```

The IIS IDS layer hasn't changed much from the first incarnation earlier in this chapter. Instead of receiving a single IP address sent over an unencrypted network connection, the IDS layer listens for and accepts connections that contain public key cryptostreams that need to be deciphered in order to be used. The code shown here also expects any number of IP addresses sent one per line as ASCII text. The key pair is loaded using the FromXmlString method. It implements the inverse ICryptoTransform object that expects 128 byte input ciphertext blocks and transforms them into 117 byte (or less) output plaintext blocks, without having to remove feedback introduced during encryption since this example asymmetric block cipher is operating in ECB mode where feedback is not added through an initialization vector.

In this code example and in the previous one you saw RSA keys encoded as ASCII text in XML. The <Modulus> and <Exponent> represent the public key while the private key consists of <P>, <Q>, <DP>, <DQ>, <InverseQ>, and <D>. The entire XML encoded key pair is unwieldy and you aren't likely to spend time memorizing your keys. This creates a practical security concern because you need a safe and reliable place to store the bytes that represent your keys. Smart cards are a good solution to this problem, although anything but memorization pretty much means you have to protect your key storage at all costs. Printing out keys and typing them in manually each time they're needed isn't a bad solution, it just takes a while to key in all those characters. The most common solution is to encrypt long keys with a shorter key, such as a password. Deciding how to manage keys and passwords is an important part of a comprehensive security policy.

Each time data is decrypted using an asymmetric private key, it can be reencrypted using a different asymmetric public key for which the encrypting computer has no matching private key. In this way information can move from one system to another with maximum protection against theft by preventing a computer that is finished processing data from

further accessing that data. To prevent attacks on automated systems that must have real-time access to decryption keys requires all of the tools of information security. There must be an incident response plan for the scenario where decryption keys are stolen or discovered through cryptanalysis, because there's a chance that either incident may happen. Whenever possible, automated decryption should not be part of secure applications. Decryption should occur only at the request of a human user who is in possession of the decryption key.

As you can see, there are special security considerations inherent to load balancing server farms and gardens. In addition to helping prevent DoS conditions during attacks, and generally making Web applications more responsive and scalable, a network load balancing (NLB) server farm opens up the potential for new types of attack between nodes and against administrative locations that are afforded special trust in order to enable them to manage farms. Restrictive security policy can automatically block traffic based on IP address when malicious activity is detected as an attack countermeasure if occasional disruptions to non-malicious users are acceptable. Honeypots, nodes that exist only to attract attacks so that attackers reveal themselves as such before they are given a chance to probe your server farm for vulnerabilities are also valuable tools for managing security in a server farm.

The most important security tool your server farm has is its intrusion detection system, or IDS, and it is only as good as its ability to consolidate security context gathered by other devices, its own security monitoring, and honeypots. Every installation of IIS on the NLB nodes of a server farm needs to have an IDS layer that enables as much of the security context information gathered by an IDS to be used when making decisions about whether or not to process requests from certain clients and if so whether those client requests should be afforded reduced trust because of warning signs picked up by the IDS. Only an IDS layer built into IIS can enable this type of context-sensitive application security. In addition, when SSL encryption is used to secure communications with clients, only the IIS node itself knows the content of each request, so only an IDS layer that gets involved in request processing after decryption has occurred has the ability to watch for suspicious or obviously malicious requests and patterns of client behavior that are indicative of an attack.

Application-specific server farm-compatible cryptography is also an important consideration. Public key cryptostreams act as lightweight digital signatures for trust management in a server farm through the use of one-way asymmetric encryption. Applications that run inside the farm for the purpose of moving sensitive data through the farm to or from other networks or the Internet receive a substantial security benefit through the use of asymmetric ciphers in bulk encryption. There are many circumstances in which a server farm node will never again need the ability to read certain data that it received over the network or produced locally as part of a data processing routine. In such circumstances, symmetric encryption may be less desirable than asymmetric encryption due to the fact that the server farm node, or any malicious code that may be running on it, can decrypt ciphertext produced using a symmetric encryption algorithm as long as it still has the secret encryption key. Because server farms are complex distributed computing networks used by large numbers of clients and they typically act as information processing gateways they are more likely to be attacked. Therefore server farms must implement

better security measures than other computer networks and as an administrator or programmer who works on a farm you must anticipate penetration by an intruder.

Through careful planning and a server-farm compatible Web application design, you can catch intruders quickly and contain the damage they are capable of doing.

Chapter 4: Platform Security

Historically, much of the damage caused by security holes in IIS deployments came as a direct result of misconfigured security in the Windows platform. From the perspective of defending against unwanted code execution, the default settings of the Windows platform make optimistic assumptions about network security. For example, the default configuration settings in Windows assume that the typical installation will prefer to grant access to whatever useful services the platform is capable of providing rather than forcing the computer owner to explicitly authorize each service on each interface. It isn't surprising that many IIS deployments would still have security flaws related to Windows platform configuration even after the latest service pack and hotfixes have been installed.

This is a by-product of the fact that most computer owners need software to solve specific problems. While everyone also needs data security, only software used for providing high-risk services in a high-risk environment such as a public data network or a military installation needs to make paranoid, pessimistic assumptions about network insecurity that increase administrative burdens and inconvenience the typical user without good reason.

Unfortunately this means that even IIS, which are arguably intended to be deployed on their namesake public network, the Internet, exist by default on a computing platform that trusts the network not to do bad things. Perhaps more importantly, IIS exist on a platform that, historically, has not been asked by its owners to be capable of withstanding malicious onslaughts and repel the full range of infosec threats. As a result, the Windows operating system trusts itself not to have security bugs that could turn useful services that it provides by default into weapons used by attackers to facilitate penetration or cause service disruptions. Eliminating security bugs in operating system code is important, but security bugs are sometimes subjective. A security bug that prevents you from deploying IIS as a hosting platform for other people's code has no impact if you only use IIS for hosting your own code and no malicious third party code ever executes on your IIS box. A bug that allows malicious code to improperly elevate its privileges, intercept user account credentials, or install a Trojan to do these things later is a vulnerability only when malicious code is able to execute in the first place. Likewise, a security bug that makes IIS unsafe to deploy on an Internet-connected computer without a firewall and other third-party security solutions is only a problem if you insist on deploying IIS on the Internet without these extra protections. With common sense as a caveat, and an understanding that the industry norm is to deploy services for the Internet only in conjunction with specialized network security devices such as firewalls, it's important to harden your IIS boxes to make them nearly-impenetrable to begin with just in case they end up directly exposed to malicious attacks.

Hardening your IIS platform starts by disabling unnecessary pieces of your OS installation. Unnecessary features in the platform that underlies an IIS deployment represent complex unknowns for the security foundation of IIS. Most IIS installations exist in the context of vanilla Windows OS installations that haven't been purged of features that aren't essential in part because operating systems tend to provide many more services than most owners actually need. To ensure the viability of a common operating system

platform into which customers deploy a mix of code from a variety of vendors with the reasonable expectation that code from any vendor will function properly when installed, Microsoft adopted and maintains the viewpoint that core OS features should always be present on any Windows box. Prior to Windows 2000, any OS feature that might not be present on a target Windows box could be freely redistributed by other vendors whose software required that feature. Starting with Windows 2000 the notion of redistributable OS components disappeared in order to avoid the undesirable and problematic version conflicts and security concerns that came with third-party distribution of OS components. Now, only Microsoft decides what goes in the Windows OS and customers are expected to leave all the pieces in place even if they aren't used. Starting with Windows 2000 you can't even remove most OS files because they are automatically replaced by force if they ever get deleted or altered because of the ill-conceived and badly-implemented Windows File Protection (WFP) feature.

WFP is supposed to keep the Windows platform rich and functional and support the existence and financial viability of a large base of independent software vendors, many of whom would not survive if customers who purchase their products can't use them out of the box due to the possibility that OS code needed for add-on software products to work properly may not be present as expected on the target computer. WFP was supposed to eliminate customer support problems caused by corrupted or missing system binaries, provide a measure of protection against rootkit style Trojans, and rescue everyone from DLL Hell. Windows File Protection is a complete failure, technically. That Microsoft's best efforts to solve the problems caused by their own insistence that Microsoft themselves and a bevy of third-party Independent Software Vendors (ISV) must have arbitrary and unrestricted access to your microprocessors should ultimately fail miserably is no great surprise. Restricted access or access governed by your own security policy and a subset of the Windows OS binaries that you choose to trust would result in the death of many ISVs as well as cause harm to Microsoft's business practice of forcing Windows users to accept everything Microsoft wants them to use and making everything programmable so that ISVs can come in right away and try to sell users more software that they don't need and shouldn't want.

To properly secure the IIS platform, you must disable OS features that aren't hardened and purge any features that aren't explicitly necessary. Deleting many OS files from the box completely isn't possible in Windows 2000 and later without disabling WFP, so the best you can do is just disable the features they implement. Infosec best practices are to physically remove all code that is not actively used by a box. However, this runs counter to conventional wisdom concerning the value of a standardized platform, therefore the Windows server platform doesn't permit the computer owner to exercise control of the OS code that is resident on the box by default. Further, the only options for disabling features are those provided by Microsoft as additional features, which creates something of a paradox. Everything is on and active by default when you install any Windows server operating system, and Microsoft doesn't provide you with the option of disabling features unless Microsoft agrees that it should be possible to disable them. This has been, and continues to be, a fundamental problem with Windows platform security, and it's not something that will change any time soon. As a result, your only option if you wish to disable features that you don't trust but for which Microsoft has not yet released another feature to allow you to disable the first feature is to convince Microsoft Security that your

idea for a way to disable a certain feature that you don't want is worthy of development efforts that will lead to a new feature provided by a future service pack.

After you've hardened Windows platform security to your satisfaction, the next step is to make sure that the application hosting infrastructure provided by IIS is adequately secured for your needs. In addition to configuring security parameters for application hosting under IIS and designing trustworthy application security features, you must make sure that the scripts you deploy as the core application content served by IIS are properly security-hardened. Secure scripting requires both an awareness of threats and vulnerabilities inherent to Web applications and a concerted effort to avoid sloppy development practices that create flaws that are otherwise easily avoided. Microsoft Active Server Pages (ASP) and ASP.NET enable secure Web applications when the proper techniques are employed to ensure server-side script security. The next three chapters delve deeply into these topics, beginning with the complexity of securing an operating system designed to be programmable at every level and act as a foundation for a type of network computing where every node in the network is designed to be client, server, peer, and host of application services or code delivered to the node over the network.

Windows Platform Architecture

The Windows operating system is the result of decisions to make an operating system that is programmable in every respect and at every level and to encourage third party developers to write software that is programmable in every respect and at every level. The benefits that result from these decisions are widely believed to outweigh the risks. Among other benefits, this type of pervasive programmability gives users more features when and if they decide they need them. It also enables every piece of hardware and software to work with every other piece of hardware and software, even if it is necessary to inject an adapter module between two incompatible pieces to make them work together. When every piece is programmable, adapting the functionality of any piece to new and previously unforeseen uses becomes simple. Third party developers can, in principle, build code modules that customize any OS feature without access to the source code used by Microsoft to create the feature.

While Windows and its programmability philosophy have provided computer owners, and third party developers, with choice and new potential around every corner, the very things that give the platform these characteristics also make it exceptionally difficult to secure. There exists in every operating system module and third party software product deployed under the Windows platform the potential for malicious configurations and uses. In addition, the operating system caters to programmers more than to infosec professionals and end users who would prefer an operating system that is more configurable, more stable, more secure, easier to understand, and less programmable. Code written by Windows programmers and by Microsoft itself is far more complex than it needs to be simply to get a limited computing task done; every bit of it also complies with interfaces designed to enable other programmers to interact with and use the code, and even script or reprogram it, as the programmer sees fit. Rather than forcing users to follow a rigid, fixed procedure to get from point A to point B while working with software, Windows takes great pains to make sure that users can define the procedure they will follow and even change that procedure, and thereby alter which code executes, each time Windows-

based software is used. The technical features that enable this versatility and programmability are:

- An event-driven message-oriented subsystem underlying all programs
- Component Object Model (COM) interface definitions for compiled code
- ActiveX and its support for scripting of COM objects
- Distributed COM (DCOM) and seamless integration of network-based code
- Application Programming Interfaces (APIs) for every OS layer and feature

When combined with public key cryptography-based digital signatures for code signing to automatically authenticate trustworthy executable code and for encryption that facilitates trustworthy communications with remote network nodes, Microsoft Windows provides transparent and automatic location independence for code execution that enables every computer to be client, server, network peer, and hosting platform in a highly-automated and completely programmable distributed computing environment. This architecture sounds very good to Windows programmers, but these features bring a risk of misuse and unanticipated configurations that can be manipulated for malicious purposes, often automatically through remote control. And these features do not change the bottom line that you own and control your computers and the software they execute. At least until a malicious third party takes that control away from you by exploiting some security vulnerability, and within the constraint of being forced to run certain OS features in order for Microsoft Windows and its network services to function properly. Ownership gives you the right, and the obligation, to take whatever measures you deem necessary to protect your property, and prevent its misappropriation as a weapon that can harm others. To the extent that Microsoft Windows' security foundations fight against you by preventing you from configuring the optimal level of pessimism and restrictions on feature and code availability on your IIS box, the struggle to secure the IIS platform can be quite frustrating. The key is to be aware of the risks you must take in order to own a box that runs a Microsoft Windows server OS and that connects to the Internet.

Security Foundations of Windows

Microsoft DOS and Windows 3.1 had a single-process, single-threaded, flat, 16-bit memory model that distinguished between ring 0 (protected mode) and ring 3 (user mode) protection levels defined by the Intel processor architecture but otherwise these 16-bit platforms offered no mechanisms to prevent malicious code from accessing or modifying other code or data loaded into RAM. For the most part, any code was just as privileged as any other code. When user mode code attempted to access protected mode code without the help of a microprocessor security gate, a general protection fault (GPF) would result. Windows NT and Windows 9x changed the Windows platform significantly by introducing a multi-threaded, multi-process 32-bit memory model and additional security layers on top of the low-level microprocessor protection levels and pushed more of the operating system into protected mode, which had previously served mostly to distinguish between device drivers and everything else, the user-mode application software. Windows 2000 patched security holes in the 32-bit Windows platform and added new features for secure networking, replacing some of the more desperately problematic and terminally-insecure legacy Windows networking code. Windows XP and the .NET Server Family now provide not just more features that might improve security if they're used properly (and if Microsoft's programmers hit the nearly-impossible target of shipping bug-

free code) but also a security-optimized operating system architecture free of certain security problems that impacted even Windows 2000.

Windows' programmable platform was originally designed around the needs of consumers and small- to medium-sized businesses where trustworthy isolated computer networks or no networks at all were the norm. What those consumers and businesses needed most, originally, was as much software as possible as quickly as possible. Windows was originally designed to make sure that programmers could write and ship software easily and quickly so that consumers and businesses would become dependent on the features enabled by software and therefore want more software and more computers. This architecture sounded very good to Windows programmers. While their need for software was being satisfied, Windows customers realized, thanks to the Internet and the spread of multimedia computers, that they also needed digital content and computer networking. Microsoft responded by embedding computer networking deeply into the operating system and making digital content nearly indistinguishable from software so that Windows programmers could create content and software for computer networks, too. This new architecture sounded very good to Windows programmers. Meanwhile, consumers and businesses began to realize that they also needed security. And in classic Microsoft fashion, everything Windows consists of is being retooled in order to provide security without changing the core philosophy behind a common operating system platform that is highly-programmable and feature-rich. The result is complexity that increases faster than anyone can understand it, and that leads to unavoidable security holes.

Security holes caused by too much complexity aren't unique to Windows. Any computing platform that purposefully makes itself more programmable and more complex, while software developers attempt to control the security implications of this increased complexity faster than the complexity increases will encounter exactly the same problems. There is an important infosec lesson to learn from the experience of others in this respect: unnecessary complexity reduces security. As you build or manage applications under IIS remember the KISS principle: Keep It Simple, Stupid! Windows itself arguably discards this principle in favor of the MIPS principle: Make It Programmable and Ship It!

However, all Windows platform programmability rests on a small number of control points that function as limiting factors for complexity. By concentrating on these limiting factors, and when you know where to find the Windows platform's control points, the complexity that looks unmanageable on the surface can easily be monitored and its risk factors mitigated.

Windows Registry and NTFS

One of the most important control points for security in Windows is the Registry. In Windows NT and Windows 2000 you use `regedit.exe` or `regedt32.exe` to edit the Registry manually. The difference between the two is substantial: `regedit.exe` can't modify Registry permissions or add new MULTI type values to Registry keys. On the other hand, `regedt32.exe` isn't as user-friendly as `regedit.exe`. Windows XP and Windows .NET Server combine the two Registry editors into a single enhanced `regedit.exe`. The screen shots in this chapter use only `regedit.exe`, and the standard user account permissions

configuration popup window that you've seen a million times just didn't seem important enough to include a picture of, so you can use regedt32.exe or the XP/.NET regedit.exe to see this window for yourself if you'd like. Choose Permissions from the Security menu to see the user interface Windows provides for configuring permissions on any securable object as you read on.

The Registry is not simply a persistent storage location for configuration data that Windows and application software needs access to each time they launch. Windows also uses the Registry as it exists at run-time to manage in-memory data structures that are important for system operation and security. These transient parts of the Registry may never be written to hive files on disk, but they are accessed by pieces of the OS such as the kernel by way of the same Registry API functions as are normally used to access Registry keys and values that live in persistent file hives. The HKEY_USERS and HKEY_CURRENT_USER registry keys are good examples of transient parts of the registry that are populated and used at runtime. An API function, LoadUserProfile, gives unprivileged Windows software the ability to populate HKEY_USERS with the information necessary to carry out operations programmatically using the security context and user logon profile of a particular interactive user account. After calling LoadUserProfile, application software will typically call CreateProcessAsUser to kick off code execution in the context of the specified interactive user account, with privileges and preferences unique to the user.

Persistent hive files are located in the System32\Config directory. To back up these files, including the SAM hive which is not visible to any user except LocalSystem at runtime you can boot into the Recovery Console in Windows 2000 and .NET Server. The Recovery Console gives you access to the NTFS disk partitions in a simplified shell so that you can make simple changes to the filesystem without booting the OS. Files that are normally locked and inaccessible except to the service process that holds the files in memory can be copied or replaced from the Recovery Console. If you can't take a server out of service in order to run the Recovery Console, or if you're using Windows NT, you can still backup the SAM hive using a simple trick that allows the Registry editor to perform a full backup of this hive in spite of the fact that it is locked for exclusive access by the LocalSystem account. With the Task Scheduler service running and configured in its default security context of LocalSystem, execute the following shell command:

```
AT [current time plus 1 minute] /INTERACTIVE "regedit.exe"
```

Replace [current time plus 1 minute] in your AT command line with a time string such as 12:01pm if the current time happens to be 12:00 noon exactly. The Task Scheduler will execute the regedit.exe command using the LocalSystem security context and display the program interactively for you to use. The entire contents of the Registry can be accessed and the export or backup menu selections used to save Registry data to file. Needless to say, the Task Scheduler should not allow just anyone to schedule tasks to be executed in the LocalSystem security context.

When originally designed, Windows loaded the Registry files into memory as part of the kernel mode paged pool memory region. The paged pool region can be swapped out to the system virtual memory paging file as needed during system operation to free up space in physical RAM memory which is the only memory the system can use at runtime

for computing operations. The non-paged pool kernel mode memory is never paged to disk, so it's available in physical RAM at all times and to all kernel-mode callers. The non-paged pool is where all stack space for kernel-mode processes is allocated. Paged pool memory pages are swapped out to disk when they are unused for a period of time, and Registry data is used by nearly every application, resulting in the fact that Registry data constantly occupies a portion of the paged pool.

Knowledge Base Articles Q182086 "How to Clear the Windows NT Paging File at Shutdown" and Q295919 "How to Clear the Paging File When You Use the Sysprep Tool Prior to Imaging Windows 2000" detail features that enable a Windows Server to automatically erase the virtual memory paging swap file when the OS shuts down in order to make it more difficult for sensitive information contained in the swap file to be copied. This feature doesn't prevent computer forensics tools from reading erased information off the hard drive, however, because the swap file data is not overwritten on disk, a normal file delete occurs.

In Windows 2000 and prior OS versions, the layout of the kernel's virtual address space limited the size of the paged pool to about 160 MB. As the Registry grows larger and larger, and more and more applications depend on it as a database service of sorts, the amount of memory left in the paged pool for every other dynamic kernel memory allocation decreases. In Windows XP and .NET Server the Registry is moved out of the paged pool and into the non-paged pool where the kernel cache manager manages on-demand mapping of 256 KB regions into and out of memory.

For more information about the Registry and its storage in paged pool kernel memory in legacy versions of Windows consult Knowledge Base Article Q247904 "How to Configure the Paged Address Pool and System Page Table Entry Memory Areas," Knowledge Base Article Q94993 "INFO: Global Quota for Registry Data," and new 64-bit memory architecture described in Knowledge Base Article Q294418 "Comparison of 32-Bit and 64-Bit Memory Architecture"

Like the Windows Registry, NTFS disk partitions provide a variety of security features that are critical to preserve system integrity. By itself, the Registry can prevent unauthorized users from invoking all sorts of components and services because it provides security configuration features that aren't filesystem-dependent. But without security at the filesystem level the Windows platform has no way to prevent access to files that contain important operating system configuration, such as the Registry hive files, or program files located in paths that are already configured as trusted by certain Registry settings. It is easy, for example, for any user who has access to the filesystem to replace with a malicious rootkit Trojan any program file that corresponds to a service configured in the Registry to execute under the LocalSystem security context unless NTFS permissions prevent the user from modifying the program file that corresponds to the privileged service. Even when NTFS is used, anyone with physical access to the hard drive can connect it to another computer where they have Administrative privileges and thereby circumvent NTFS permissions. Only by using a facility such as the Encrypting File System (EFS) feature of NTFS under Windows 2000 and later can files be protected both at runtime through NTFS permissions and at all other times, even when physical security is compromised, through the use of strong encryption. Unfortunately, EFS isn't designed to support encrypting OS files because it operates using public key pairs assigned to

each SID. If OS binary files are encrypted using EFS, only the user who encrypted them will be able to decrypt them, and the computer won't function properly for any other user.

Primary and Impersonation Access Tokens

The smallest security primitive unit in the Windows platform is the thread of execution whose current stack frame represents the point of origin and security context of any attempt to access any securable object. There is no security other than that provided by encryption and physical access controls without a thread of execution to give meaning to security information and to enforce security policy primitives by loading OS code into process memory and calling into OS code already running in other processes created by the OS. Each thread, and every process, is given an access token when it is created by the operating system through a call to the `CreateProcess` or `CreateThread` API functions.

When a thread in an existing process creates a new thread, the default access token assigned to the new thread corresponds to the current access token of the calling thread. When a thread creates a new process, the default access token assigned to the new process corresponds to the access token assigned previously to the process which owns the thread that calls `CreateProcess`. The token assigned to a process when it is first created becomes the primary token of that process and all of the threads it owns. A thread can switch its security context and effective access token from its primary token to an arbitrary access token, called an impersonation token, that defines a different security context by calling `SetThreadToken` and passing as a parameter any access token or by calling `ImpersonateLoggedOnUser` and passing as a parameter an access token that represents a currently logged-on user. When an impersonating thread is done using the impersonation token it can call `RevertToSelf` to switch back to the primary token. When a thread is impersonating a security context other than its primary token and the thread calls `CreateProcess`, the primary token is used to assign the new process its primary access token, not the impersonation token.

The default behavior of `CreateProcess` and its use by COM produced a security flaw under IIS 4 on Windows NT whereby an in-process application running with a primary access token of `LocalSystem` but an effective impersonation token equal to an authenticated Web site user or `IUSR_MachineName` can elevate its privileges by using `CreateObject` to launch an out-of-process COM server object without the help of Microsoft Transaction Server. The new process created by COM as the host process for the COM server is assigned the primary token of the process that owns the calling thread, which in the case of an in-process Web application under IIS 4 is `inetinfo.exe` and therefore the primary token represents the `LocalSystem` security context. If a thread wishes to create a new process using its current impersonation token, it must call `CreateProcessAsUser` or `CreateProcessWithLogonW` instead of `CreateProcess`. COM+ in Windows 2000 added a facility known as cloaking to preserve the impersonation token across process boundaries when an impersonating thread calls into an out-of-process COM+ object.

Knowledge Base article Q281837 entitled "INFO: COM EXE Servers Run in SYSTEM Context When Called from IIS" details the impact on Windows NT security of out-of-process COM servers launched from in-process IIS applications

Security Identifiers

To understand the contents of an access token you need to be familiar with the other security data structures used internally by the Windows platform. The most important element of an access token is its user account security identifier (SID). Every distinct user account and every group that represents a security principal for granting or restricting access to programs and data in Windows is assigned a unique SID when it is created. The structure of an SID is a hierarchy of unique numbers. When represented textually rather than as a binary data structure, an SID begins with the letter S to indicate that the data that follows represents the elements of a security identifier. Next, an SID contains a variable number of fields separated by dashes starting with its data structure format Revision Number, and then followed by a 48-bit authority identifier, I. Finally an SID contains unique relative identifiers issued by each subauthority that enable the SID to uniquely identify a security principal within the security domain identified by I and within the specified subauthorities.

S-Revision Number-I-Subauthority Relative Identifiers

Windows provides numerous static SID definitions that have built-in meaning called well-known SIDs. There are universal well-known SIDs and well-known SIDs that only pertain to Windows NT or later. There are also well-known SIDs that have meaning only relative to a Windows domain controller if the box is a domain member. When you have a need to know well-known SIDs, they're easy to find in Windows SDK documentation. To see real-world SIDs your Windows box uses to identify a real user, look in the Registry under HKEY_USERS.

In addition to the user account SID, an access token contains a list of SIDs that identify group membership for the user account SID. There is one group SID for each Windows group, and each group SID is unique just as each user account SID is unique. The privileges granted to each group as well as the privileges granted to the user account are merged into a privilege set that becomes part of the access token. In the event that the access token was created by a user other than the one represented by the token's user account SID, the owner/creator's SID is also included as part of the token. The source of the access token can be an important security consideration for any code that reviews the token and makes an access determination based on its contents. In addition to storing the owner's SID, each token also keeps a record of where it came from or the source of its existence. A token also specifies the default Discretionary Access Control List (DACL) for any securable object created by the security context when an explicit security descriptor is not specified for the new object. Whereas access tokens define security context and relate to security principals, securable objects in the Windows platform are responsible for defining access controls that must be observed and satisfied whenever a security context attempts to access such an object. These access restrictions are defined through security descriptors, DACLs, SACLs, and ACEs.

Security Descriptors, DACLs, SACLs, and ACEs

All securable objects in the Windows platform have associated security information known as a security descriptor. An object that is not securable does not have a security descriptor, for example a file stored on a FAT filesystem rather than NTFS does not store a security descriptor along with its other data, and a securable object that has a null security

descriptor explicitly grants access to any security context. Security descriptors originate from explicit access restrictions configured for the object by its creator/owner, or by another user security context that has control over security configuration such as the Administrator account, as well as from implicit access restrictions assigned to objects through security descriptor inheritance or default configuration settings provided by Windows when it was first installed.

Each securable object exposes its security descriptor through a consistent set of API functions like `GetSecurityInfo` in order to conform to a consistent security programming model. Files and folders on an NTFS disk partition, Registry keys, processes, threads, synchronization primitives like mutexes, events, and semaphores are just some examples of securable objects. Process and thread objects are special securable objects that also have a security context (access token) associated with them because these objects map to OS primitive processes and threads that can initiate attempts to access other securable objects. Every object in the Active Directory is also securable, and the Microsoft .NET Framework transforms every function call into a securable object as well. At the level of coding support for these new features into the OS platform and preserving a consistent meaning for any Windows security context with all other securable objects, programmers continue to rely on the same security programming API that has been tested and debugged in real-world situations since Windows NT was first developed. This doesn't mean that there aren't still bugs that have not yet been found and fixed or that new code will automatically be more secure just because it's built on old code that is believed to be hardened and proven secure in practice, but it may be a reason to have more confidence in new code. The old limiting factors and control points for platform security still have the final say over access controls through the use of well understood security primitives.

A thread's primary access token represents its limiting factor for access to securable objects on the Windows platform. Any code that owns a thread that has an access token for the `LocalSystem` can potentially access any and all securable objects that exist on the box. Even if a thread has a less privileged impersonation token in effect, there are a variety of ways for a thread to `RevertToSelf` even when that privilege is supposed to be restricted in the impersonated security context. The only way to be certain of the restrictions placed on a thread is to assign the thread a primary access token that is itself restricted. This means that the security context that each process executes within is a far more important consideration for platform security than whether or not threads are supposedly being assigned restricted impersonation tokens.

The Windows platform implements a standardized access check procedure whenever a particular security context as defined by an access token from a calling thread requests access to a securable object that has a particular security descriptor. Inside a security descriptor is an SID indicating the owner of the securable object and a Discretionary Access Control List (DACL) which contains Access Control Entries (ACEs) specifying user and group access permissions. The owner of an object has discretionary control (by way of the DACL) over the list of ACEs that exist to grant or deny access to other security contexts. Each ACE present in the DACL lists a particular SID and a particular configuration of permissions that are explicitly granted or denied. The SID can be that of an individual user account or it can correspond to a group. By matching the SIDs contained within the thread's access token against the SIDs contained within the security

descriptor, Windows is able to determine whether the security context should be granted or denied access to the securable object. All ACEs are examined in the order they appear in the ACL to determine access rights of the calling thread. ACEs that apply to the caller's security context aggregate such that a right denied the members of a group but explicitly granted to the user account whose SID appears in the access token will be granted to the calling thread. This is due to the order in which ACEs are normally listed in an ACL, known as the canonical order. Unusual orderings are possible, though not as a result of using GUI interfaces in Windows programs that allow you to configure permissions for securable objects. The GUI interfaces for permissions all construct ACLs using the canonical ordering established for ACEs in order to provide a consistent interpretation of access permissions during a standardized access check.

Component Object Model

From a practical information security viewpoint, only highly-skilled and experienced programmers with special security training should build compiled code to expose services for automation and computer networking because of the relative difficulty of security-hardening such software compared to software that exposes no services. However, Microsoft Component Object Model (COM) is designed around a different security viewpoint that places modular code design by as many programmers as possible as a higher priority than preventing code that has not been security-hardened from executing automatically in the context of a module that other code can interact with programmatically. To compensate for the security problems created by enabling programmers to write COM components that aren't properly hardened but that can nonetheless be invoked by callers programmatically in the context of in-process or out-of-process servers, Microsoft .NET forces components to implement basic security features and defaults code to a more hardened state. You can and should view legacy COM programming tools prior to .NET managed code development tools as appropriate for use today only if the developer who uses the legacy tools knows how write secure code, and takes the time to do so. It's important to understand that COM doesn't change the way executable code is formulated and loaded into memory from the filesystem (or from other memory) and it doesn't change the technical details of process, thread, file, and other operating system primitives. COM simply defines a high-level interface for executable binary code to communicate with and identify other code across well-defined logical boundaries that map to each of the possible combinations of operating system primitive boundaries. Security bugs in COM, bugs in operating system features that use COM, bugs in third party code modules that use COM, and bugs in IIS/COM integration don't cause most IIS vulnerabilities: insecure platform configurations do. Further, insecure platform configurations don't go away by installing service packs and hotfixes to repair specific security bugs in COM, IIS, or OS binary modules.

Disabling Distributed COM

COM is extended by Distributed COM (DCOM) which builds on the COM foundation to enable COM compliant code to call other COM compliant code on other computers across the network. Just as COM is designed to allow compliant code to communicate across process boundaries, DCOM allows code to treat the network as the functional equivalent of a process boundary. Every COM compliant program is automatically DCOM compliant because configuration settings for each COM component determine whether

the component executes locally or remotely. DCOM applications can explicitly request, and even require, remote execution rather than local execution, as an optional feature for DCOM programmers. DCOM is on by default in any Windows Server operating system installation and it uses the Remote Procedure Call (RPC) facility, derived from the Distributed Computing Environment (DCE) standard, which is also on by default. These network services represent a significant vulnerability when COM, NTFS, and other operating system features and interfaces with a security impact haven't been properly locked down and hardened.

To disable DCOM you can use the DCOM Configuration Utility (DCOMCNFG.EXE) as shown in Figure 4-1. This utility is also useful for configuring DCOM application security settings if you need to leave DCOM enabled for some reason. Default permissions and network transports for remote component execution are also configurable using the DCOMCNFG.EXE program. Every setting that DCOMCNFG.EXE allows you to define for DCOM on the platform can be defined directly through the Registry editor, but certain settings such as DACL and SACL binary values are much easier to construct with help from this utility.

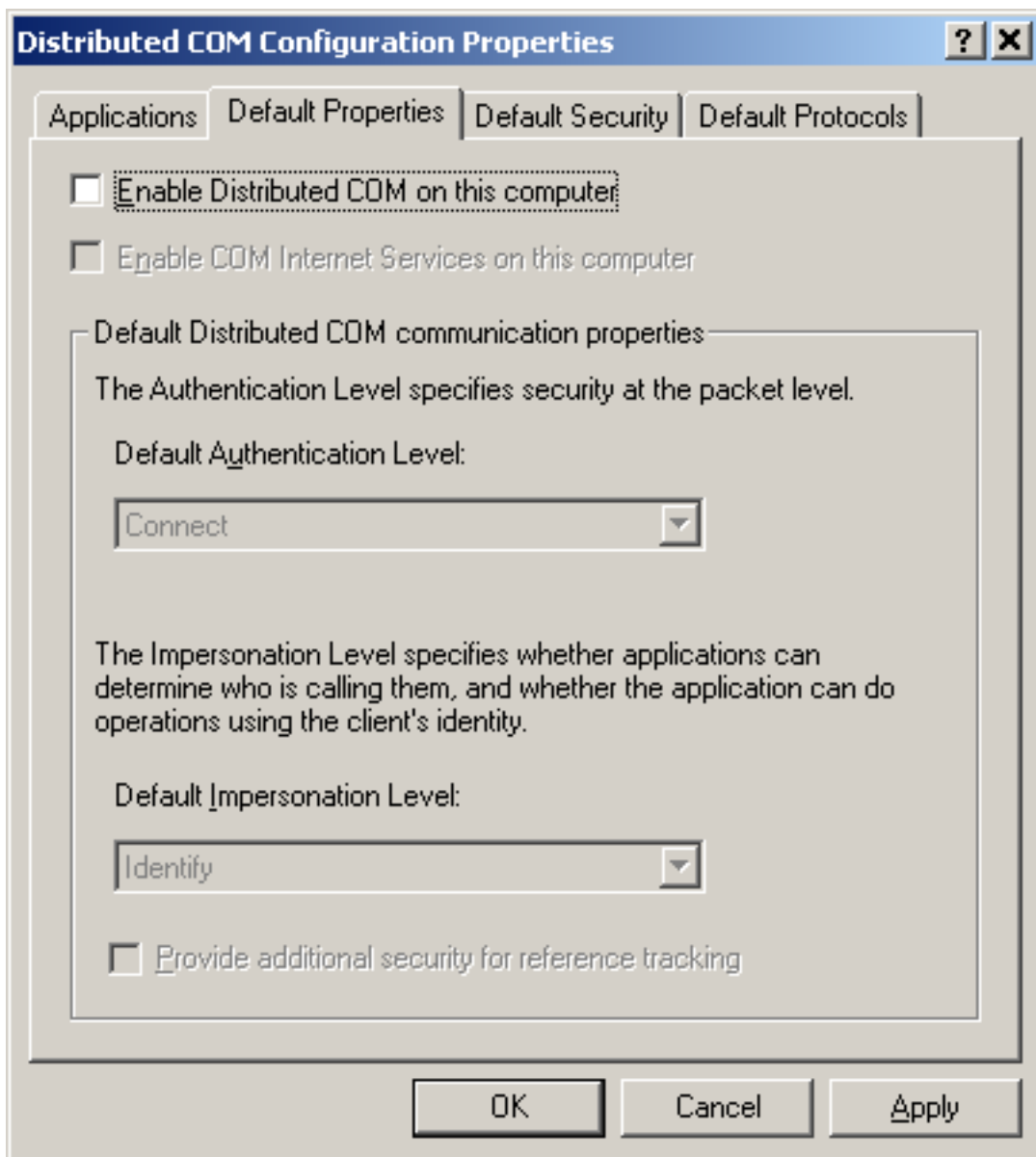


Figure 4-1: Turn Off DCOM Using DCOMCNFG.EXE

To understand the difference between DCOM, COM 1.0, COM+, and Microsoft Transaction Server (MTS) beneath the superficial difference that COM+ is the successor to COM 1.0 and MTS, you need to understand the way that COM components are registered and used on the platform. DCOM still provides remote execution for COM+ components, including the new ability called cloaking that allows a thread's impersonation token to be marshaled across a process or a network node boundary and reestablished for the thread that carries out remote execution on behalf of the caller. The essential elements of registration and use of COM compliant code in the Windows platform are described in the following sections. As securable objects at both the file and Registry key level, every COM component offers multiple control and auditing points with conventional DACLs and SACLs. This provides a platform standard mechanism for application launch and call security even when the programmer who codes the COM component includes no additional security features.

Application ID, Class ID, and Program ID

Every COM-compliant component is assigned a unique Class ID (CLSID) by its developer that is set in the Registry as part of the SOFTWARE hive when the component is installed and registered. CLSID values are typically produced through the use of a developer tool that creates a Globally Unique Identifier (GUID) according to the DCE standard algorithm. One such tool is guidgen.exe, a utility that Microsoft provides as part of the Windows SDK. Every component is also assigned a Program ID (ProgID) that is a short name for the class. ProgID is created arbitrarily by the programmer without the use of any ID generating tool. The ProgID can serve as a version independent identifier for applications that want to instantiate COM classes in such a way that instances of the newest code are always created when components are versioned and newer code becomes available on the system. As securable objects, every CLSID and ProgID Registry key and value carries with it a DACL that defines access permissions and a SACL that defines auditing policy. CLSID entries are contained under the following Registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID

Whereas ProgID entries are found in the parent Registry key named Classes. Figure 4-2 shows the CLSID and ProgID entries added to the Registry for the IIS Web Application Manager (WAM) COM component System32\inet_srv\wam.dll. The DACL on the CLSID provides platform-level permission restrictions on access to the services provided by the WAM at runtime. The only way a malicious attacker can gain access to the functionality provided by the wam.dll when it is loaded in-process by inetinfo.exe is to take control of inetinfo.exe. Any unprivileged process can potentially load the wam.dll into memory and call its functions, which probably would not give malicious code the ability to escalate its privileges. A much easier avenue of attack would be to replace wam.dll with one of malicious design that exposes some sort of interface for the attacker to inject commands or machine code into inetinfo.exe or through which the attacker can extract data out of inetinfo.exe at run-time. As long as the NTFS DACL on the wam.dll file prevents unauthorized tampering by any unprivileged user security context, tampering isn't a big concern while the OS is running. Windows will also keep a file lock on any EXE or DLL file that is currently loaded into memory by a process, such as wam.dll while it is loaded into inetinfo.exe, further protecting a COM class module binary file like wam.dll from tampering on-the-fly. But during system startup and shutdown, and at any time that the NTFS partition might be mounted on a different computer by a malicious third party, such files are subject to tampering. Windows 2000 and later OS versions include Windows File Protection, a facility that will automatically replace any OS binary that is changed or deleted at run-time. While this feature is useful, it isn't designed to be fool-proof.

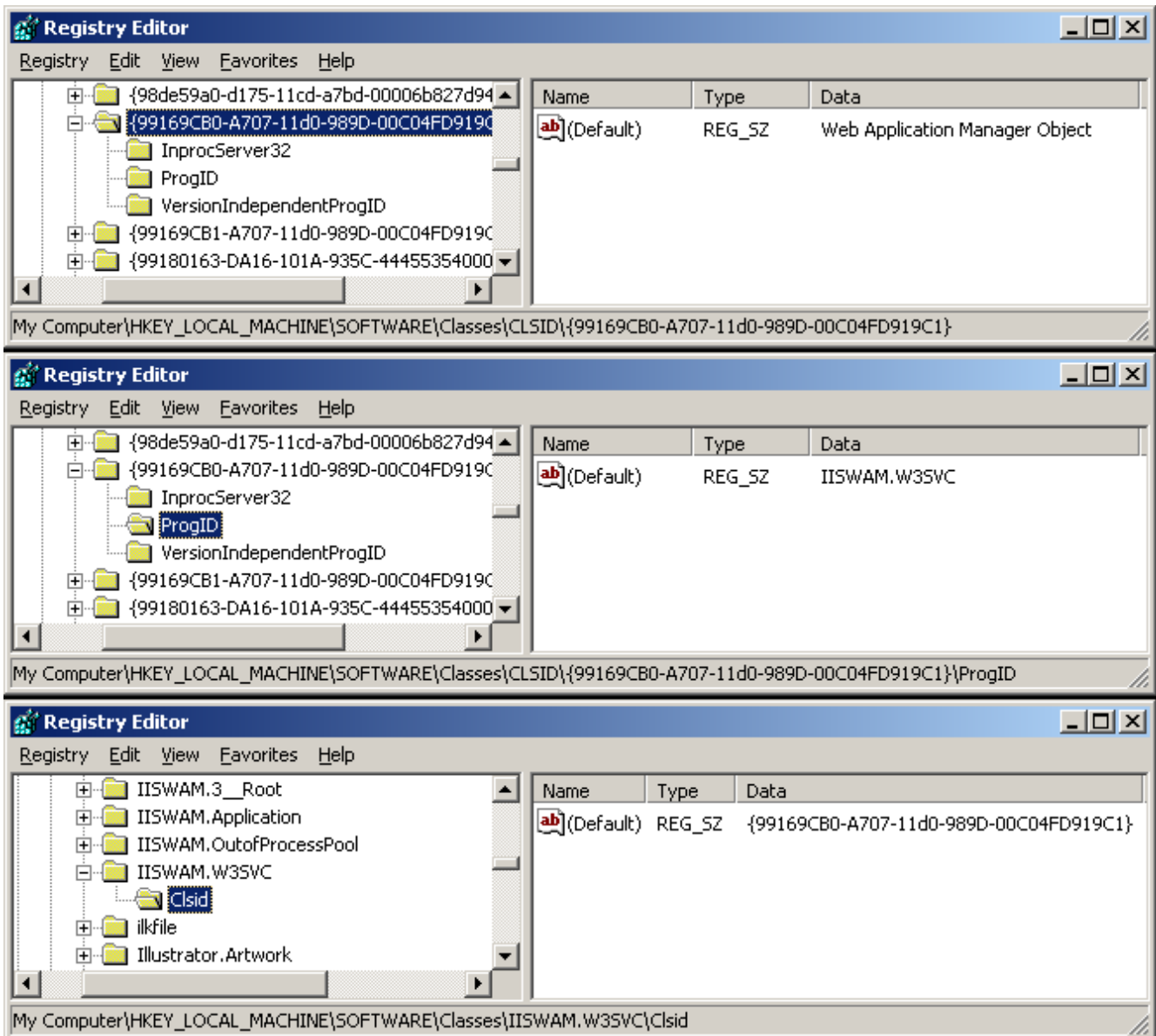


Figure 4-2: COM Classes Are Registered by CLSID and Also by ProgID

Every COM class registered on the system can also have an optional Application ID (AppID). The AppID of any in-process COM server, that is any COM class whose configuration in the Registry includes an `InprocServer32` key and value, associates that in-process component with a configuration set that includes access and launch permissions and the name of the executable file that hosts the component. Components that specify the same AppID share the same executable host process which can be either a `LocalService` or a `DIISurrogate`. The default `DIISurrogate` provided by COM+ is named `DIISHost.exe` and this is the same surrogate that hosts WAM applications configured in IIS 5. IIS 4 uses Microsoft Transaction Server as its application surrogate host process. IIS 6 replaces the COM+ `DIISHost.exe` default surrogate with its process isolation mode feature and the `w3wp.exe` worker process. Figure 4-3 shows the `IISADMIN` service process CLSID entry and its corresponding AppID. By configuring `IISADMIN` with a CLSID and AppID in this way, Microsoft enabled DCOM integration and an additional layer of permissions for accessing and launching the service.

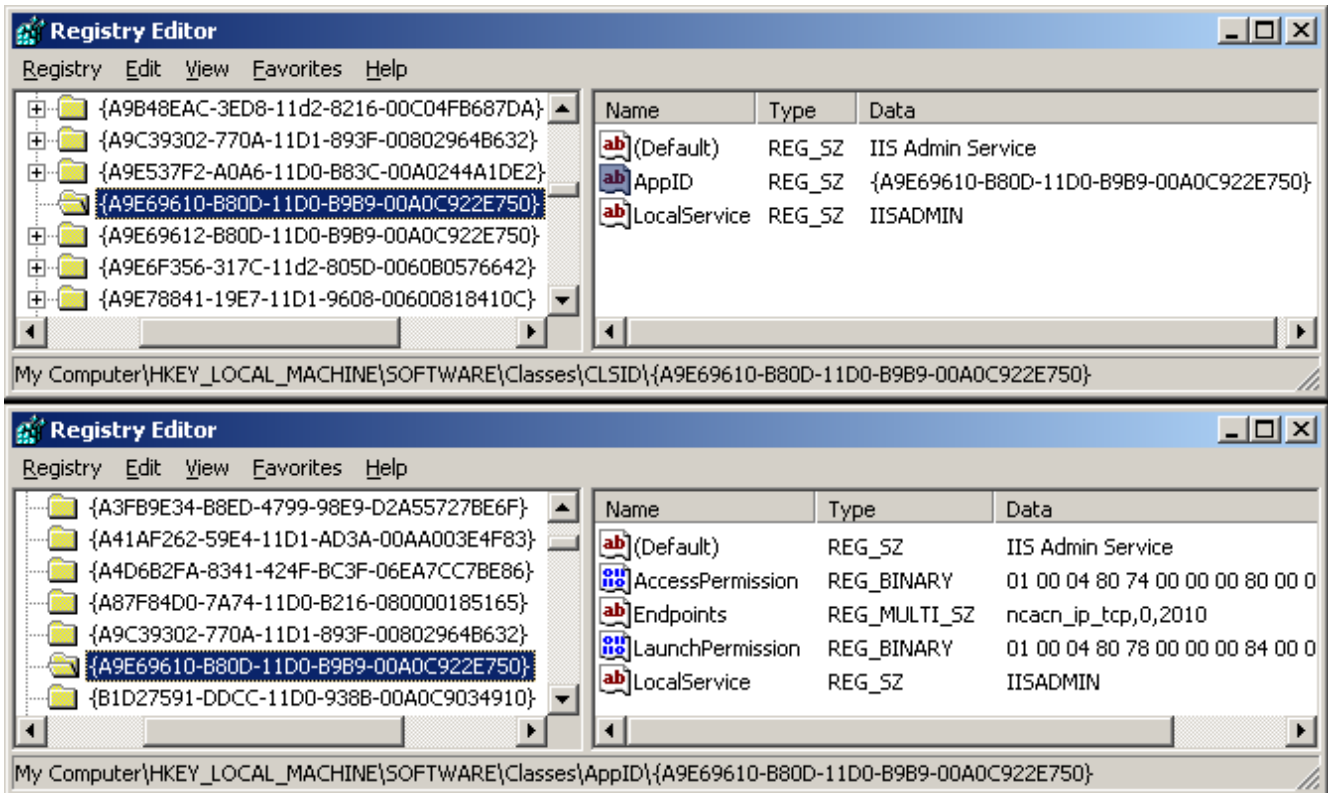


Figure 4-3: IISADMIN is Configured in the Registry as a COM Class and a DCOM Application

When DCOM is disabled as shown previously in this chapter, certain AppID features that facilitate remoting are also disabled in addition to disabling the DCOM features that allow the local computer to receive and process remoting calls that originate elsewhere on the network. The DCOM AppID as a mechanism to force in-process COM components to host out-of-process instead as though they were written to be DCOM compliant and automatically marshal parameters in out-of-process method calls is also disabled. This gives you an assurance that your in-process components will actually execute in-process as you expect them to do. Each AppID appears under the following Registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Classes\AppID

Knowledge Base articles Q246054 and Q216051 give more information about the use of DCOMCNFG.EXE to make changes to and create AppID Registry keys. Knowledge Base article Q198891 gives more information about the potential for AppID to be used as a mechanism to force COM components to execute out-of-process transparently to the caller and without the caller's knowledge and consent.

Type Libraries

Some COM classes are designed to be programmed or scripted by third parties more easily than others. When a programmer intends to use a COM class only for internal functionality within their own application, they will typically not provide any documentation to third parties about the interfaces and data types used by the COM class. The

programmer already knows these things, and they don't intend for other programmers to do anything with the COM class that would require sharing this knowledge. When a COM class is intended to be used by third party programmers, the developer typically provides a Type Library (TypeLib) along with the class. For the sake of simplicity, a developer can link a TypeLib with the object code of the COM class as a resource in the resulting DLL or EXE file. Alternatively the developer can separate the TypeLib from the compiled object code for the COM class and distribute a .TLB file to third-party developers. It's important to note that whether a developer intends for third parties to make use of a COM class or not is irrelevant to the capability of the third party to do so. Very few COM classes are designed with the idea that the interface they expose for COM client threads to call into must be security hardened and should require authentication before they agree to execute on behalf of a caller. As a result, nearly all COM classes are vulnerable to attack or misappropriation by malicious client code. Any client that can invoke COM objects can potentially exploit them to do harm. This has particularly critical security implications within the context of Web server application development for IIS as a result of the integration of COM support into any server-side application logic such as ASP script or ISAPI modules. A COM class that includes a TypeLib as a resource linked with its executable object code will typically register the existence of that resource in a Registry key named TypeLib. Figure 4-4 shows Registry keys for the ADSI namespace provider IIS namespace that provides programmatic access to the metabase along with its TypeLib Registry key. Access to adsiis.dll and its hosted COM component can be controlled by setting the DACL on the component's CLSID Registry key separately from the settings that control access to the TypeLib for development tools that simplify programming or management with ADSI by displaying type and interface information to the developer or administrator.

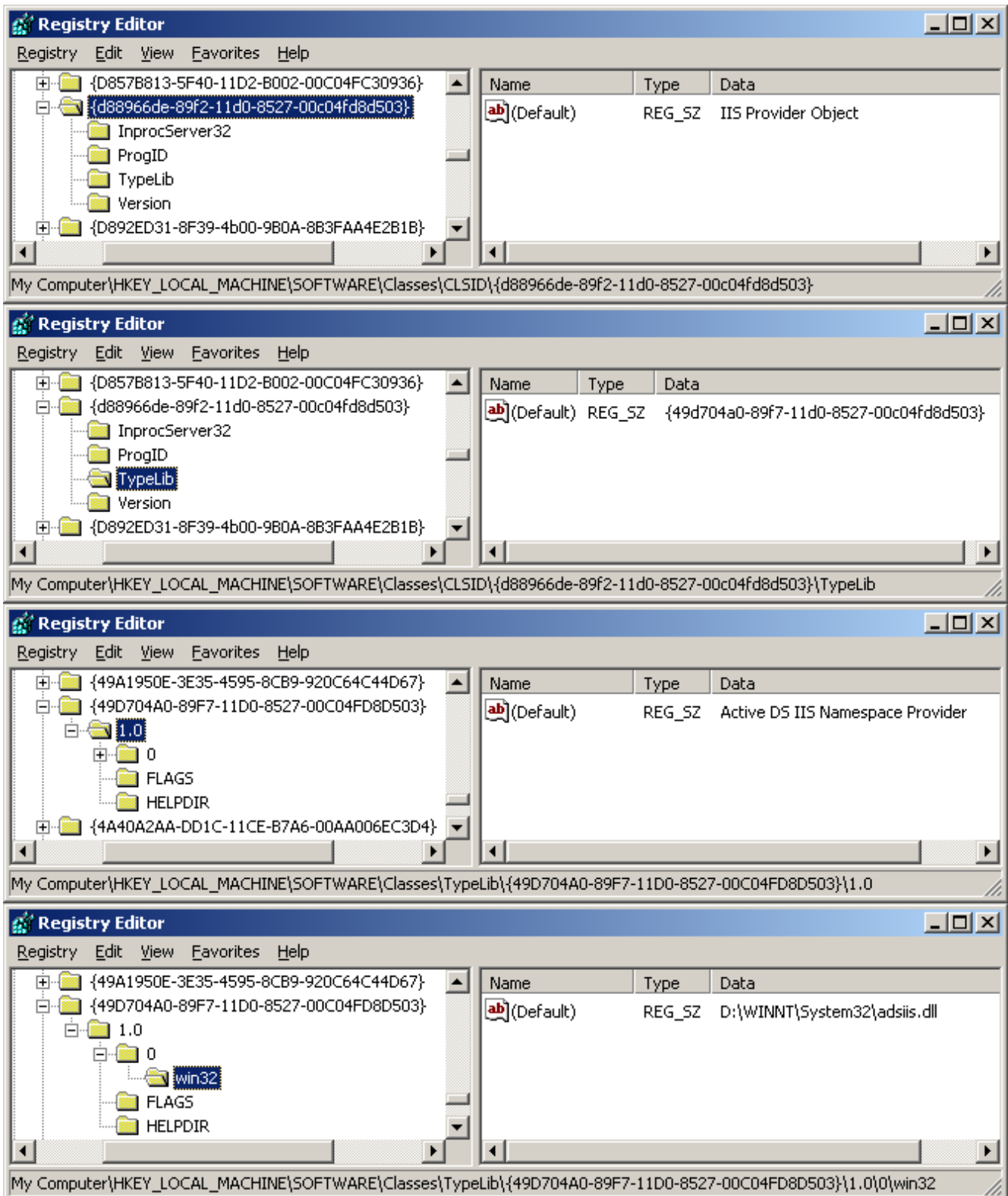


Figure 4-4: The IIS Namespace Provider for IIS (Metabase) ADSI Access Includes a TypeLib

By default Microsoft Windows binds every protocol, service, and network client interface to every network adapter. This is in keeping with the idea that the typical deployment will want to trust the network and make use of it whenever possible. For networks that are connected to the Internet, this idea is backwards. What you really want is to distrust the network by default, in both directions and on every interface. Due to the Microsoft Windows legacy and its design primarily for use in trustworthy networks, it is impossible to configure Windows networking for optimal security on any conventional network interface. This means the network interface adapter card has to provide additional protections, or you have to deploy a firewall as extra protection, which still leaves the computer vulnerable to attack from points on the network that exist behind the firewall and therefore aren't filtered or blocked.

The Client for Microsoft Networks must be installed in order for any IIS 5.0 services other than IISADMIN to start. Knowledge Base article Q243008 explains that the services hosted by IIS 5.0 depend upon the RPC Locator Service which is configured along with several other services such as LANMan Workstation when Client for Microsoft Networks is added to the system's network configuration. While Client for Microsoft Networks must be installed on any Windows 2000 box that runs IIS, it does not have to be active in order for IIS and their hosted applications to function normally. Under Windows 2000 the first, and most important, place to disable Client for Microsoft Networks is right in the network configuration window itself and you can do this immediately after installing this unnecessary (but mandatory, due to the fact that IIS won't run if the RPC Locator Service Registry entries are missing) OS feature. Simply uncheck the check box that appears next to the Client for Microsoft Networks as shown in Figure 4-5 to remove the component binding from the network interfaces represented by the adapter.

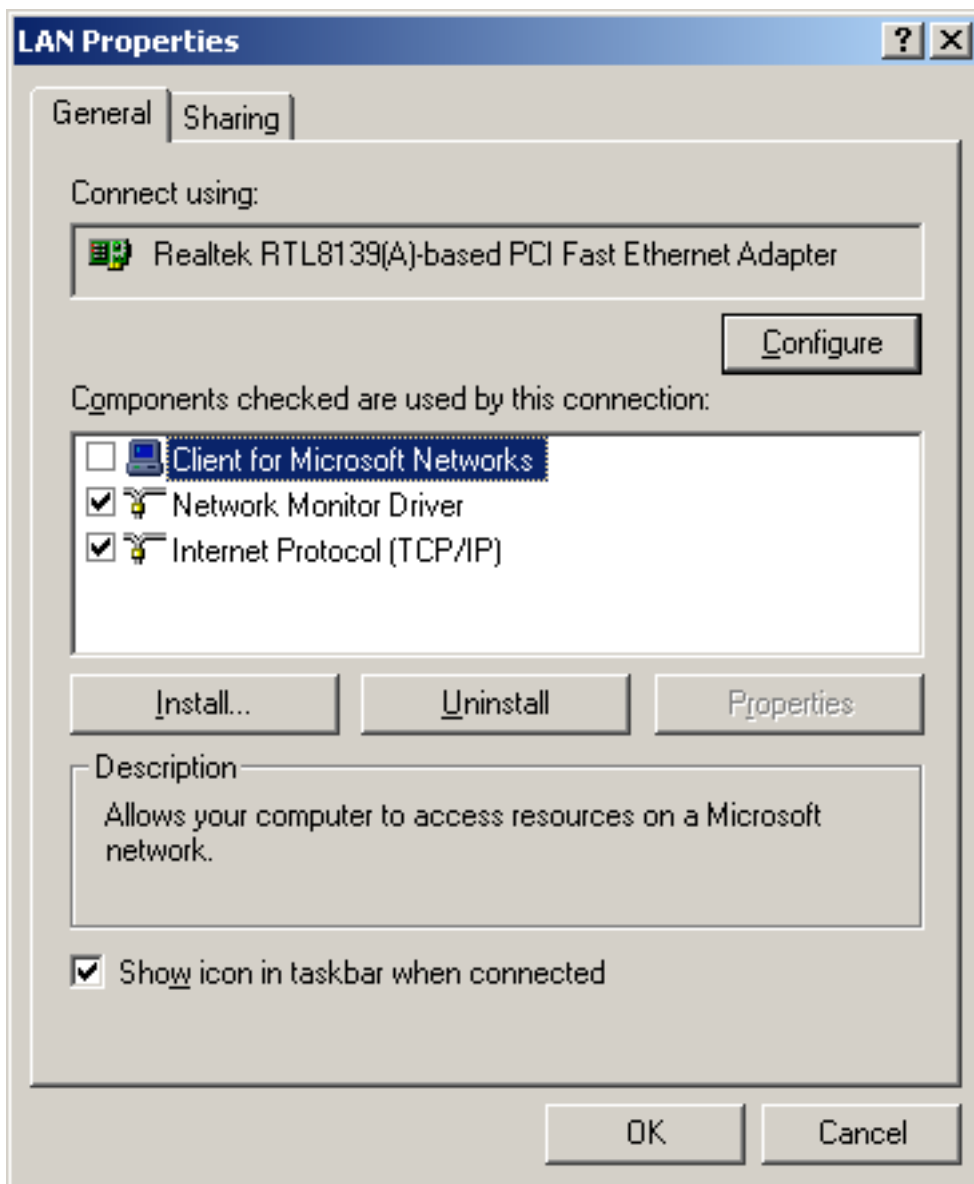


Figure 4-5: Uncheck Client for Microsoft Networks to Unbind it From The Adapter

Next you should disable each of the services and bindings added to the platform when you were forced to add Client for Microsoft Networks against your will. Do this using the Services administrative tool, which you can access through the Control Panel. You must do this in order to truly disable the client functionality because unbinding Client for Microsoft Networks only removes the NetBIOS over TCP/IP (NetBT) protocol, it does not disable Server Message Block (SMB) and the so-called “direct hosting” of SMB over TCP/IP through the NetbiosSmb device. The NetBT_Tcpip device is bound to each adapter individually, but NetbiosSmb is bound to all adapters by design. You should also be aware that file and printer sharing, if it is installed, continues to function through SMB direct hosting on the NetBT_Tcpip device unless you remove it entirely. The following services are added to the platform as a result of adding Client for Microsoft Networks to the network configuration. Each of them except RDBSS includes a Service Control Manager (SCM) that enables the service to be configured and controlled through the Services administrative tool.

Alerter
Computer Browser
Messenger
Net Logon
NT LM Security Support Provider
Redirected Drive Buffering SubSystem Driver (RDBSS)
Remote Procedure Call (RPC) Locator
Workstation (LANMAN)

None of the client services listed are required by IIS. You should set them all to Disabled in the Services administrative tool. The dependency that exists in IIS that requires Client for Microsoft Networks and its services to be installed is a ridiculous and useless one that hopefully will be deprecated in a future service pack. As a general security principle to live by, you should not configure or program servers to also have the abilities of clients if this can be avoided. When you are unable to separate these two functionalities for some reason, you should at least ensure that the network interface through which a box provides services is always different from the interface used by the box for client access to other services provided by remote network nodes. The only reason to leave Client for Microsoft Windows services running is if you've configured a second network adapter in your IIS box so that the box is multihomed and it will use the Client for Microsoft Networks services to communicate with other nodes on the second network on which the IIS box provides no services. In this case bindings and linkages for network services, protocols, and adapters must be carefully configured for a hardened multihomed deployment.

Multihomed Windows Networking

A multihomed network node is one that has multiple network adapters that are typically (though not necessarily) connected to different physical networks. When a single network adapter has multiple addresses bound to it in order to give it multiple network identities on a single interface, the node is said to be logically multihomed rather than physically multihomed. Every network node that runs TCP/IP is automatically logically multihomed in a certain sense because of the existence of the loopback interface 127.0.0.1 that the node can use to reference itself without using its designated name or routable IP address.

This is important to understand because a variety of malicious attacks are possible whereby a network node is instructed to reference itself by way of its loopback address.

The subtle security implication of this automatic logical multihoming become a little more clear when you consider that any TCP/IP network service that binds to all available interfaces (the virtual IP address 0.0.0.0) creates a service that is accessible from applications running locally by way of the automatic loopback IP address 127.0.0.1.

Multihomed configurations are a common network security technique in TCP/IP networking. Configuring a single network node with multiple IP addresses and connecting a single node to multiple networks through multiple network adapters provides a variety of security benefits and important deployment configuration options. The original firewalls

were constructed in this way using regular off-the-shelf computers before specialized firewall devices were produced by vendors. The key concept behind multihomed deployments is the careful partitioning of services and settings that pertain to each network adapter and address. Although an intruder who penetrates the security of the services exposed by a node on one interface and gains remote control of the node via one of its homes can potentially penetrate the node's other homes in a multihomed configuration, the risk of penetration is lower when a single service, such as an HTTP server, is exposed through any high-risk interface rather than a plethora of services. By controlling service bindings for each network adapter installed in a Windows Server box you can prevent services from binding to Internet-accessible interfaces that have no business being accessed from the Internet in the first place.

One of the most useful multihomed network security techniques available to you in Windows is use of the Microsoft Loopback Adapter. This special adapter is a software-only virtual adapter provided by Microsoft that acts like a regular physical network adapter from the perspective of any code that is instructed to bind to it or make use of the virtual network interface it exposes. The Loopback Adapter always references the local network node, so any network traffic sent to it goes nowhere. This virtual device that leads nowhere is similar to `/dev/null` in the Unix OS. TCP/IP can be bound to the Loopback Adapter as can the services provided by the RPC Locator so that RPC endpoints created by software that isn't very security-aware can still be made available for local RPC clients to use without exposing these RPC endpoints on the physical network where they would be a vulnerability. Figure 4-6 shows the Add/Remove Hardware Wizard adding the Microsoft Loopback Adapter. To instruct the Hardware Wizard to add this adapter you must specify add a new device/select hardware from a list, then select Network Adapters as the type and Microsoft as the manufacturer. You can't plug and play the Microsoft Loopback Adapter because it doesn't exist as a physical device.

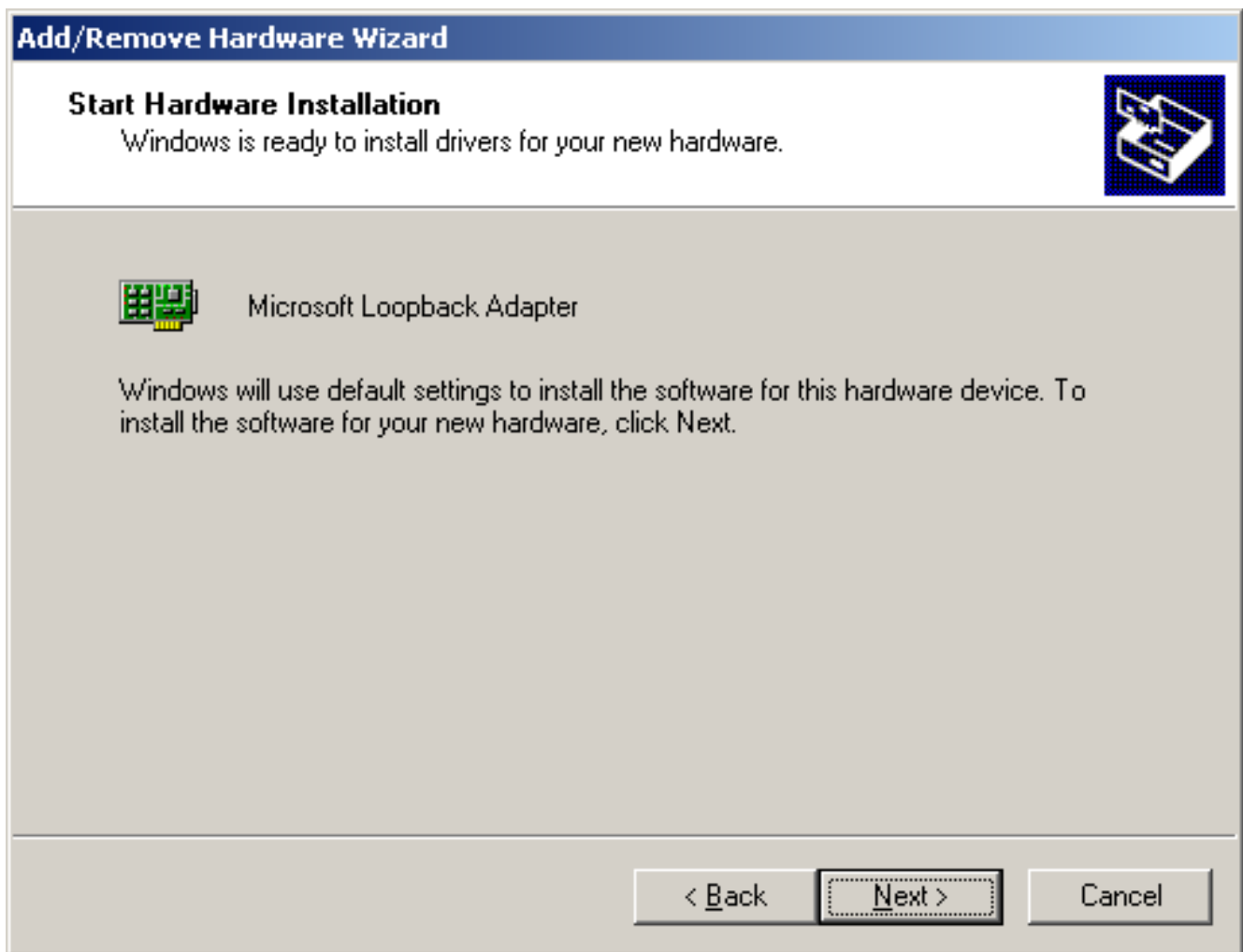


Figure 4-6: Add The Microsoft Loopback Adapter Using The Add/Remove Hardware Wizard

With the Microsoft Loopback Adapter configured on your IIS box, you can perform a variety of important platform configuration and hardening tasks beginning with a lockdown of interface linkages and service bindings. Windows is far too eager, by default, to provide services on the network. By removing certain default linkages and binding services or protocols other than TCP/IP to the Loopback Adapter, you can achieve provable security for the network configuration of your Windows box. Whether you multihome your box with a second physical network adapter and a second LAN to which to connect it so that the box truly has multiple homes or whether you rely on the Loopback Adapter as a /dev/null-style virtual home, once you've established a multihomed configuration there's some housekeeping to do.

Interface Linkage and Service Bindings

Every network adapter interface and protocol linkage and binding is configured as a virtual device driver within the Registry. By explicitly listing virtual device names and exporting those names so that other NDIS layers can bind to them by name, Windows enables every NDIS component to bind to every other NDIS component, provided that it is designed to operate in an NDIS layer that sits above or below layers that are bound to it

and to which it binds. Understanding, and controlling, these linkages and bindings, and pruning the list of bindings which by default are too aggressive to be properly hardened, is an important step that should be taken before placing any IIS box into service. If you've configured a second network adapter to multihome the Windows box or if you've installed the Microsoft Loopback Adapter, this second home is assigned a unique device driver name and it can be left in any protocol binding that needs to exist for Windows to function properly but that should under no circumstances be allowed to operate over the Internet-accessible network interface, the IIS box's primary TCP/IP home.

Linkages and bindings stored in the Registry don't translate directly to TCP/IP port bindings. But there is an indirect relationship, depending upon to which TCP/IP ports a particular network service or protocol binds, and to which virtual device drivers those protocols in turn bind, TCP/IP ports bound and set in listen mode to provide services on the network will be associated with particular interfaces. For example, NetBIOS over TCP/IP (NetBT) uses UDP port 137 for its name service, UDP port 138 for its datagram service, and TCP port 139 for its sessions. The NetBT service lives in the netbt.sys device driver, which is chained together with the legacy netbios.sys driver. NetBIOS is a software interface specification and API, not a protocol, so it will work over any transport with minor modifications. The Client for Microsoft Networks and File and Printer Sharing services in Windows 2000 and later are all designed to use NetBT rather than NetBEUI or other legacy transport as in Windows NT. This enables them to operate over routable TCP/IP in addition to legacy unroutable transports that have traditionally been associated with NetBIOS and restricted Windows Networks to small networks using unroutable protocols. NetBT is only necessary, however, for communicating with other NetBIOS-enabled computers such as legacy Windows NT and other nodes. There is no need for the NetBIOS network software API specification when Server Message Block (SMB) is "direct hosted" on TCP/IP. This is the preferred configuration for the future, as it abandons NetBIOS for host name resolution in favor of DNS. It also removes an unnecessary layer of communications code that wasn't applicable in a TCP/IP environment in the first place but was provided for backwards compatibility and ease of migration.

Direct hosted SMB enables Microsoft networking over TCP port 445 instead of NetBIOS TCP port 139. By default, both port 445 and 139 are configured for use by Windows networking services, and of the two only port 139 can be disabled. There is no way to disable port 445. This is by design. What design, and by whom, remains a mystery since there's no reason to expose port 445 at all on a box that will never communicate with other hosts using Windows networking as either a client or a server. Perhaps you will be able to disable port 445 with a new security feature provided by a future service pack for the Windows Server operating systems.

Identifying Software That Binds to TCP/IP Ports

You can see which ports are currently in use under any version of Windows using the NETSTAT -AN command. Unfortunately, under Windows NT and 2000 this command does not tell you which process is responsible for the port bindings that are currently active on the box. You can often deduce which process is most likely responsible based on port numbering and whether or not a port is bound by a process that is listening for incoming connections. But this is an imprecise way to audit your system's TCP/IP port

bindings. A third-party tool called FPORT is available free from foundstone.com that itemizes port bindings and the software that is responsible for each one.

Under Windows XP and .NET Server, the NETSTAT command accepts an additional parameter, -O, such that NETSTAT -ANO will reveal both every TCP/IP port binding and the process that owns each one. That is, unless your IIS box is infected with a malicious rootkit Trojan that alters the functionality of NETSTAT.

Table 4-1 shows representative Registry keys and values that establish NDIS layer linkage and binding on a typical Windows box. Each network adapter is identified as a class with a unique GUID value assigned during adapter device driver installation. The "NetCfgInstanceld" value lets you know what the GUID is that has been assigned to a particular adapter device. The "Export" Registry value determines the identifier used to expose a particular layer for binding by other device and protocol layers. As an example you can see in Table 4-1 that ms_netbt_smb, the Message-oriented TCP/IP Protocol (SMB session), to the Export name exposed by the configuration of Realtek RTL8139(A)-based PCI Fast Ethernet Adapter by way of the {431C0F1D-0DEA-49F0-A531-E12206FB2CFF}\Linkage Registry key. Then, ms_netbt_smb exports itself, and its explicit linkage to the Fast Ethernet adapter, under the virtual device name "\Device\NetbiosSmb". Finally, you can see that LanmanWorkstation binds to this virtual device by its name in addition to binding to the virtual device named "\Device\NetBT_Tcpip_{D1A1DFBE-E62E-4D68-8A97-1DC6BBD88B46}" which is the NetBIOS over TCP/IP binding to the TCP/IP protocol which in turn is bound to the Realtek PCI Fast Ethernet adapter.

Table 4-1: Network Interface and Protocol Linkages and Bindings

Registry Key Contents or Value
HKLM\SYSTEM\CurrentControlSet\Control\Network\ Network Configuration Subkeys
HKLM\SYSTEM\CurrentControlSet\Control\Class\{4D36E972-E325-11CE-BFC1-08002BE10318} List of Network Adapters
HKLM\SYSTEM\CurrentControlSet\Control\Class\{4D36E972-E325-11CE-BFC1-08002BE10318}\0000 "ComponentId"="pci\ven_10ec&dev_8139""DriverDesc"="Realtek RTL8139(A)-based PCI Fast Ethernet Adapter""NetCfgInstanceld"="{D1A1DFBE-E62E-4D68-8A97-1DC6BBD88B46}"
{4D36E972-E325-11CE-BFC1-08002BE10318}\0000\Linkage "Export"="\Device\{D1A1DFBE-E62E-4D68-8A97-1DC6BBD88B46}""RootDevice"="{D1A1DFBE-E62E-4D68-8A97-1DC6BBD88B46}""UpperBind"="NM""UpperBind"="Tcpip"
HKLM\SYSTEM\CurrentControlSet\Control\Network\{4D36E975-E325-11CE-BFC1-08002BE10318}\{431C0F1D-0DEA-49F0-A531-E12206FB2CFF} "ComponentId"="ms_netbt_smb""Description"="Message-oriented TCP/IP Protocol (SMB session)"
{431C0F1D-0DEA-49F0-A531-E12206FB2CFF}\Linkage "Bind"="\Device\{D1A1DFBE-E62E-4D68-8A97-1DC6BBD88B46}""Export"="\Device\NetbiosSmb"
HKLM\SYSTEM\CurrentControlSet\Services\LanmanWorkstation "Description"="Provides network connections and communications.""DisplayName"="Workstation""ImagePath"="%SystemRoot%\System32\services.exe"

```
HKLM\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\Linkage
  "Bind"="\Device\NetbiosSmb""Bind"="\Device\NetBT_Tcpip_{D1A1DFBE-E62E-4D68-
  8A97-
  1DC6BBD88B46}""Export"="\Device\LanmanWorkstation_NetbiosSmb""Export"="\Device
  \LanmanWorkstation_NetBT_Tcpip_{D1A1DFBE-E62E-4D68-8A97-1DC6BBD88B46}"
HKLM\SYSTEM\CurrentControlSet\Services\LanmanWorkstation\NetworkProvider
  "DeviceName"="\Device\LanmanRedirector""ProviderPath"="%SystemRoot%\System3
  2\ntlanman.dll"
```

By examining the Registry keys on your box that correspond to those shown in Table 4-1, you can see where network device interfaces and protocol code comes from and how it binds to other NDIS layers. Your box will use different GUIDs to identify interfaces and bindings, and the Registry keys present will vary depending upon the services, adapters, and protocols you have installed. For simplicity, just open the Registry editor and use the Find menu selection to search for the word "Linkage". You will locate each NDIS layer with a Linkage Registry key using the Find feature. Another way to search through the Registry for network-related keys is to search for occurrences of the GUID assigned to the network adapter in which you are interested.

You can also disable bindings during an unattended installation of the Windows 2000 OS by editing [Netbindings] as described in Knowledge Base Article Q263241 "How to Disable Network Bindings Using the [Netbindings] Section"

Port Filtering and Network Service Hardening

Windows 2000 and Windows .NET Server, as of this writing, do not allow you to prevent the System process from binding to port 445 TCP and UDP (used for the SMB protocol) or to the System process default TCP listening port located at or above port 1024. These ports are open by default and Microsoft does not provide a feature that allows you to close these ports, so filtering them on the interface that may be exposed directly to the Internet if your firewall fails to do its job is essential. Filter all TCP/IP ports except port 80 if you wish to allow clients access only to the IIS HTTP service on the box. Add additional ports to the list for each network service your IIS box will provide via TCP/IP over the Internet. Figure 4-7 shows the TCP/IP Filtering dialog superimposed over the Registry editor which displays the Registry settings established for TCP/IP Filtering by way of the TCPAllowedPorts, UDPAllowedPorts, and RawIPAllowedProtocols Registry values. These values exist in the Registry for each network interface bound to TCP/IP, enabling distinct port filtering settings for each interface.

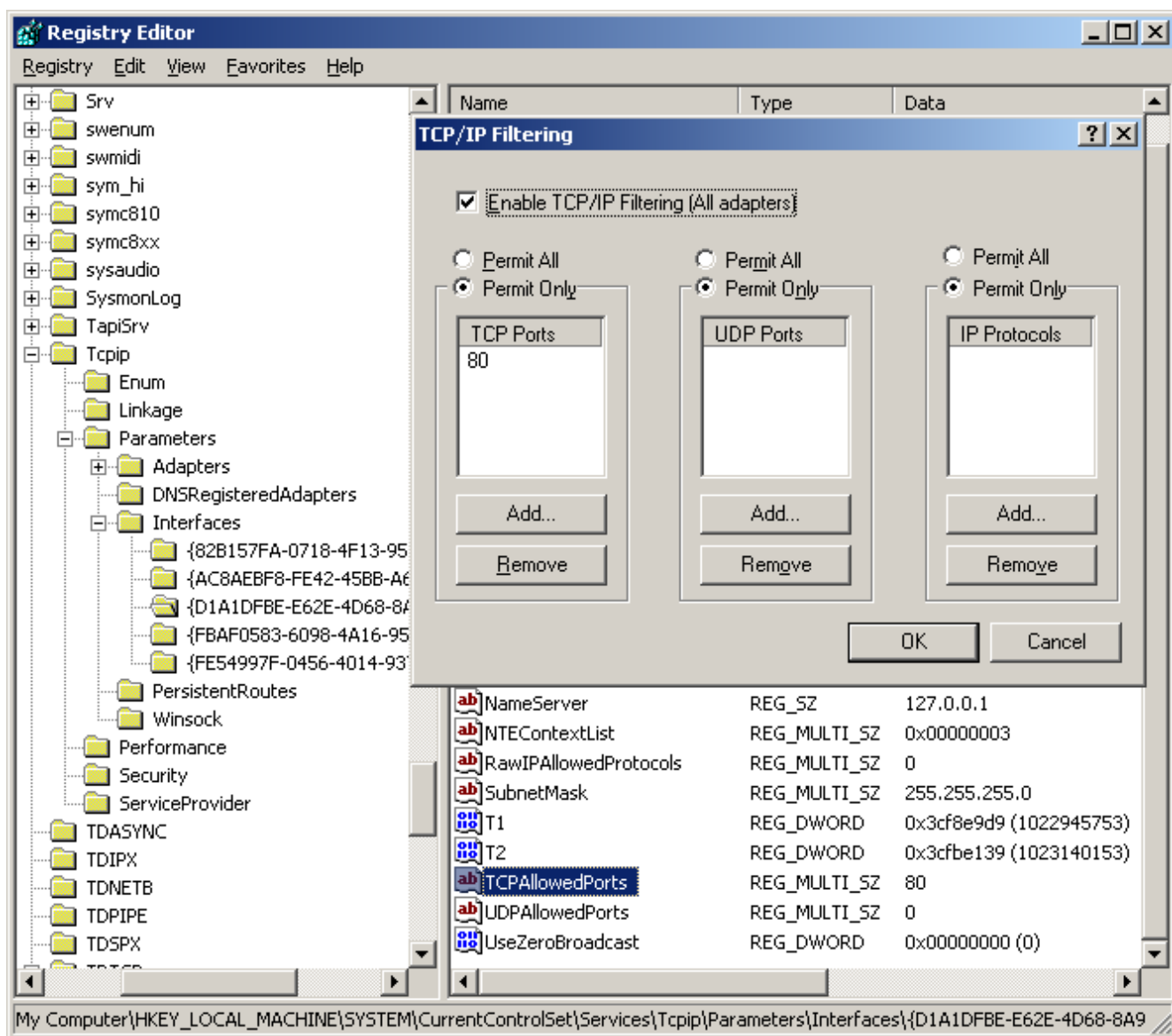


Figure 4-7: Turn on TCP/IP Filtering Through The Advanced TCP/IP Settings Window

When you enable TCP/IP Filtering, it sets a Registry value that turns on filtering for all adapters. Any interface listed under CurrentControlSet\Services\TcPIP\Parameters in the Registry that contains Allowed values like those shown in Figure 4-7 will apply the port filtering policy specified in its Registry settings. The following DWORD Registry value is set to 1 to turn on port filtering for all TCP/IP network interfaces:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\TcPIP\Parameters\EnableSecurityFilters

Even when you have nearly every port filtered, Windows still must do some work each time a packet is received on any interface. At the very least it must read the packet out of a buffer and determine whether the destination port number corresponds to any open, unfiltered port on which some network service process is bound. There is a chance that the code which reads each packet from the network adapter's buffer memory will improperly grant an attacker access to the box. The worst-case scenario is an attack

against this network device driver code that achieves a buffer overflow condition and gives the attacker the ability to execute arbitrary malicious code in a privileged security context such as the LocalSystem account. It's important to be aware that this is a possibility whenever network traffic is able to reach any network node even when there are no services available to conventional network clients through open ports with active software bindings. There is even a possibility that this will happen with your firewalls and routers. It's very difficult to prove that a network node is impervious to all possible attacks from the network unless you physically disconnect the node from the network or prevent it from attempting to process any data that arrives on any interface. Using Network Monitor (netmon.exe), however, you can observe the low-level network behavior of ports on your IIS box and see for yourself the impact that port filtering has to protect open ports from unauthorized remote access.

Monitoring SYN/ACK TCP 3-Way Handshake Packets on Open and Filtered Ports

The Transmission Control Protocol (TCP) defines a 3-way handshake for establishing a communications session with a remote TCP endpoint from a local TCP endpoint. The client is considered in the TCP session to be the network node that initiates the connection request and the server is considered to be the node that receives and acknowledges the request. Any TCP port on a box that agrees to participate in a TCP connection requested by a client is considered an open port. The client knows immediately that a microprocessor located on the remote box will attempt to process data sent to it on any open TCP port, which is the starting point for any number of attacks against the remote box. Understanding the TCP 3-way handshake is critical to understanding network security.

Network Monitor makes it easy to observe the behavior of your open and filtered ports. Port scanners and other tools can probe and interpret the state of ports automatically, but there's no better way to understand a node's ports than to see for yourself exactly what happens when packets arrive at a box and are processed by it. Figure 4-8 shows Network Monitor viewing captured packets sent to and from IIS on port 80 during a typical HTTP GET request. The first TCP packet that arrives, which is shown highlighted and labeled as captured Ethernet Frame 3, is a packet sent by the client to IIS that requests a new TCP connection by way of the Synchronize sequence numbers flag (SYN). When a SYN packet is received on an interface, the recipient is expected to read the sequence numbers provided in the packet and acknowledge the request to synchronize sequence numbers. In this case the starting sequence number is 409862 and appears in Figure 4-8 in the seq field.

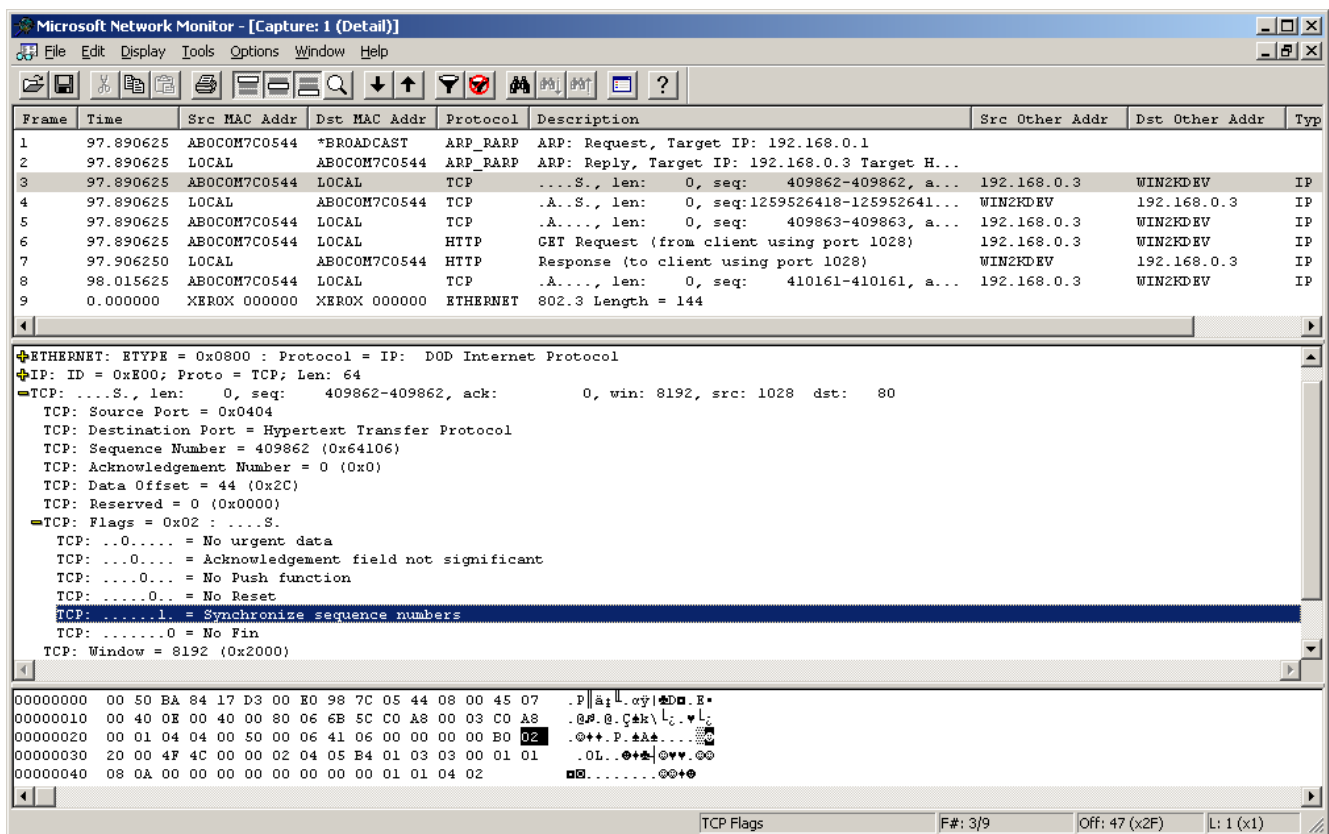


Figure 4-8: SYN Packet Arriving on Port 80 to Start an HTTP Request TCP Connection

Needless to say, if the TCP/IP device driver, which is obligated to read the sequence number field, is programmed badly so that a packet could provide a seq field that contains too much data and the device driver would attempt to fill a small memory buffer with the oversized data, a buffer overflow condition could result. The structure of most fields in IP packets is fixed-width, and the payload is always a known but variable width, so there's no good reason for a device driver programmer to get this type of thing wrong. But it's important to understand that if the programmer does get it wrong, or if a malicious device driver is installed by an attacker, the very act of a vulnerable network node receiving and attempting to read a packet could give an attacker complete control of the network node as a result of the buggy or malicious device driver. Figure 4-9 shows the acknowledgement packet (ACK) sent back to the client from the IIS box. Notice that the value 409863 is inserted into the ack field, a number one greater (the next in the specified sequence) than the sequence number synchronization starting point requested in the TCP packet shown in Figure 4-8. The TCP ACK flag is set, indicating that the acknowledgement field is significant and contains a value that should be read by the recipient. As an endpoint for a TCP connection, the IIS box must also request sequence number synchronization, so it sets the SYN flag as well in this ACK packet. The sequence numbers it provides in the seq field, 1259526418 in this case, are to be used by the client endpoint for identifying and resequencing (in the event packets arrive out of order) packets sent from the IIS box.

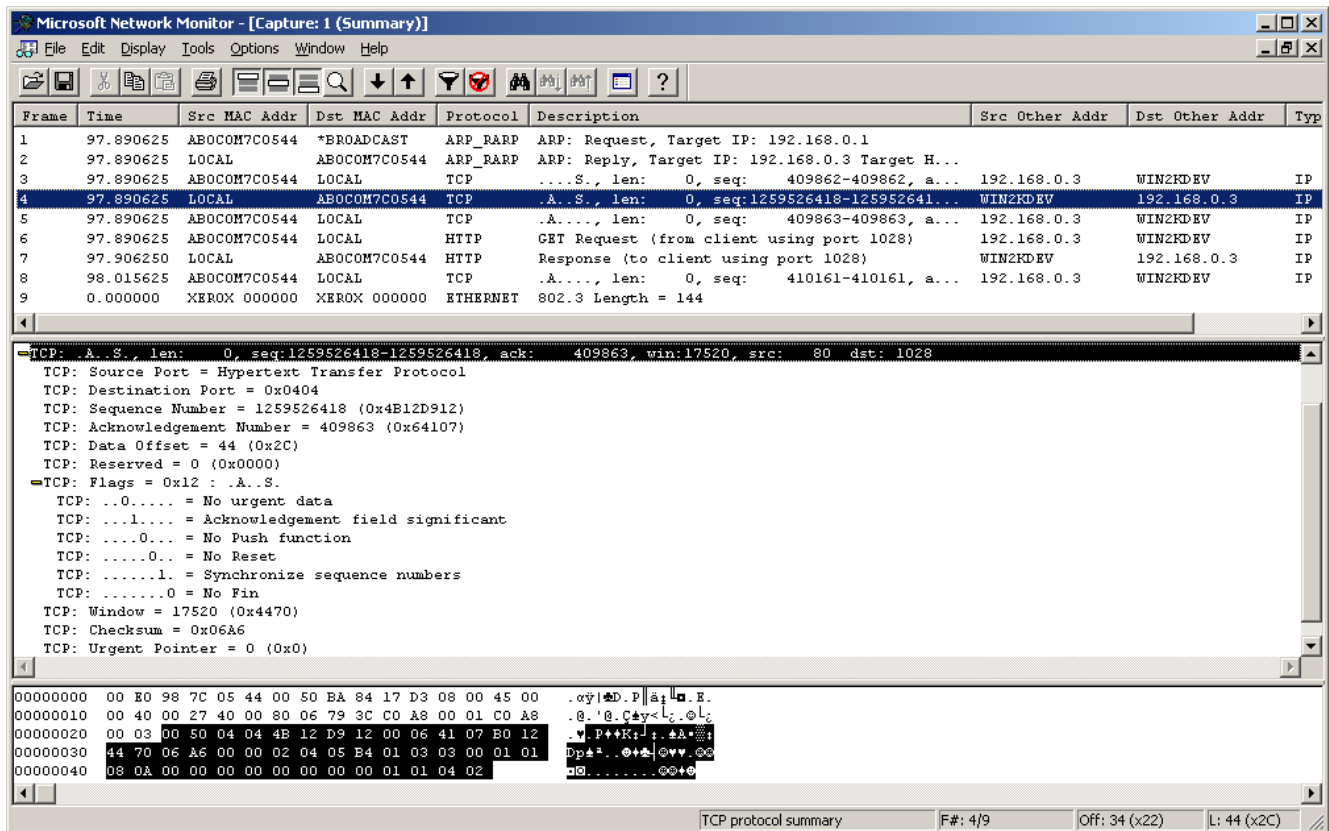


Figure 4-9: SYN ACK Packet Originating on Port 80 to Start an HTTP Request TCP Connection

Next the client sends an acknowledgement packet back to the server containing the next number in the requested sequence, 1259526419, in the ack field. The sequence number 409863 is sent by the client to match the number provided by the server in its last ACK packet. This number is one greater than the sequence number last provided by the client in its original SYN packet. Following transmission of the ACK packet shown in Figure 4-10, the client can assume that the TCP connection is open and that data can be pushed across the connection and it will be received reliably (and willingly) by the remote endpoint, the IIS box. The next packet sent by the client to the server (Frame 6) contains 409863-410161 in the seq field and once again 1259526419 in the ack field. The TCP endpoint is supposed to reject packets (or queue them for resequencing as more packets arrive) if the packets do not contain the right sequence numbers. Duplicative packet delivery shouldn't be harmful, in case some router somewhere malfunctions and sends the same packet twice, and lost packets can be detected and transmissions retried in the event of a transient routing failure. These are features of the TCP network protocol and they are the reason TCP is selected as the network protocol in many application protocols.

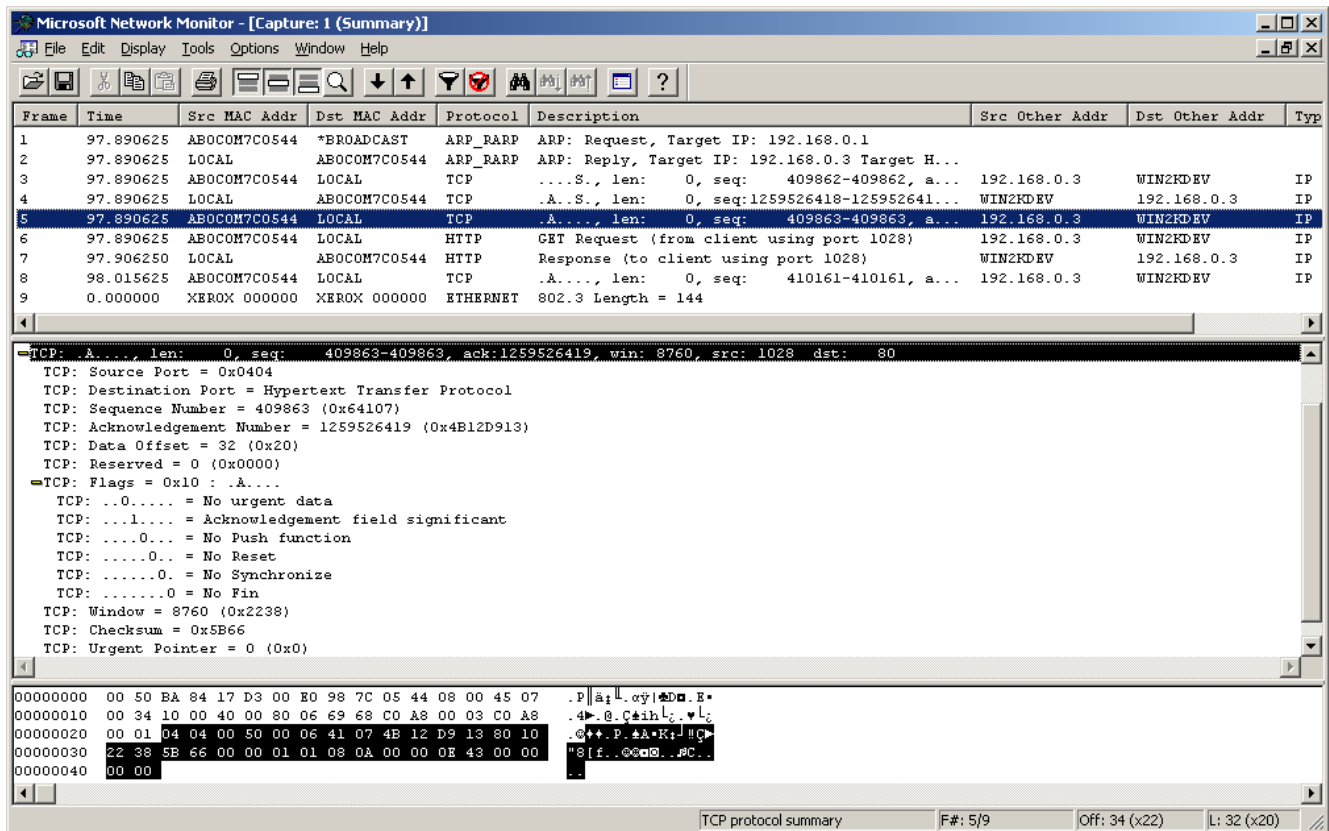


Figure 4-10: ACK Packet Arriving on Port 80 to Start an HTTP Request TCP Connection

When you turn on TCP/IP Filtering to filter ports, you instruct the device driver to refuse to relay data from packets addressed to the filtered ports on to application software that might be bound to those ports. Provided that there are no bugs in the implementation of TCP/IP port filtering provided by Windows, this makes it unnecessary to prevent applications from binding to ports where they might otherwise provide services or be attacked if not filtered. However, filtering does not prevent an attacker from discovering that a network node receives and processes packets addressed to a particular port. The attacker is unable to determine whether application services respond to packets addressed to the port, but as you can see in Figures 4-11 and 4-12 a TCP connection attempt (a SYN packet) to a filtered port will result in processing of the SYN packet and an ACK packet sent in response.

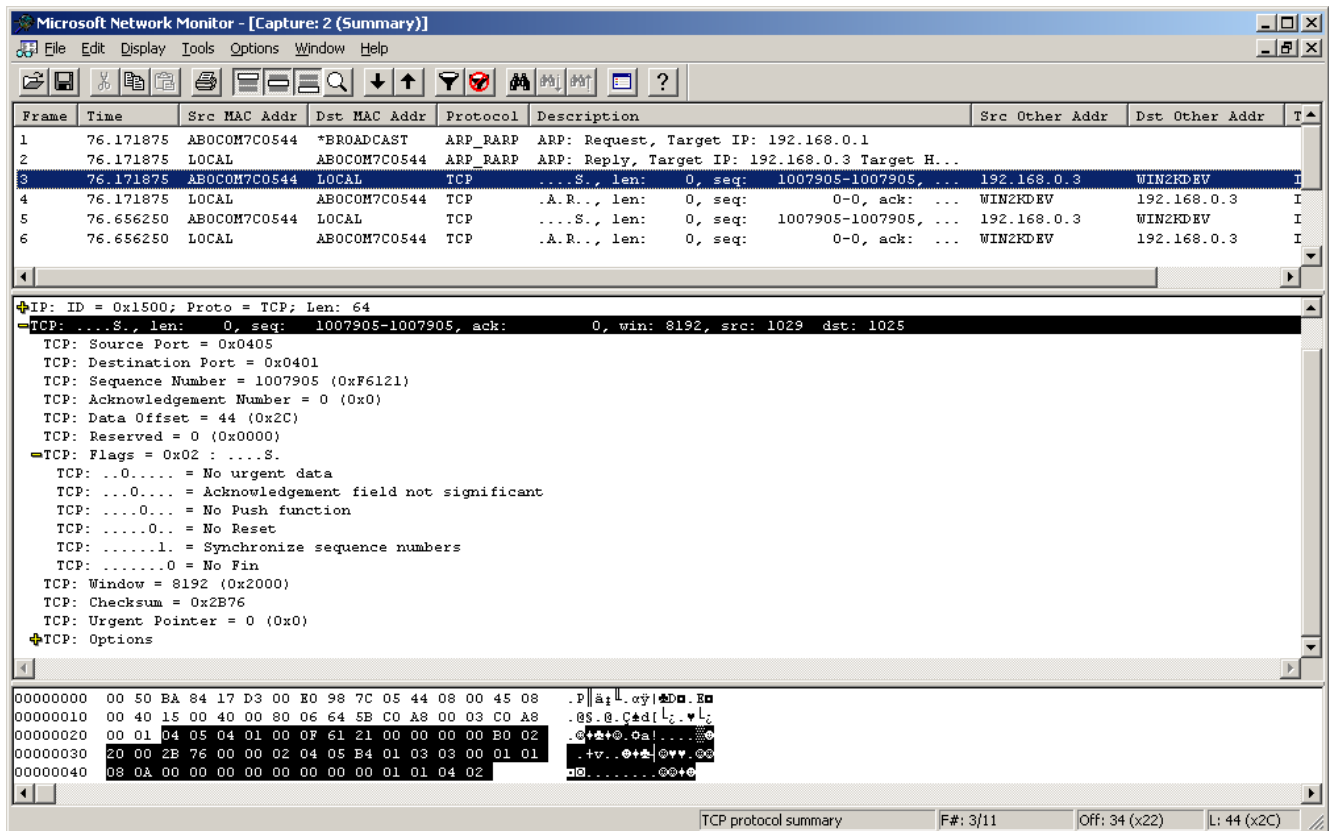


Figure 4-11: SYN Packet Arriving on Port 1025 to Attempt TCP Connection to a Filtered Port

Figure 4-12 shows that an IIS box that receives a packet addressed to a filtered port, in this case port 1025 which is filtered on a box that has TCP/IP Filtering enabled and allows only TCP port 80, always sends a response packet containing the ACK flag and the RESET flag. The RESET flag indicates to the remote endpoint that sent the SYN request that no TCP connection is being established based on synchronization of the sequence numbers it provided. The ack field contains the next number in the requested sequence, regardless, which happens to be 1007906 in this case, but unlike a TCP connection that is accepted, the ACK RESET packet tells the remote endpoint that the connection is refused. However, it doesn't indicate why the connection was refused, or whether the connection will be refused again next time, so it's common for multiple round trips to occur as TCP clients try repeatedly to establish a TCP connection. This is processing overhead that any TCP server must be prepared to service, and Denial of Service attacks are possible simply by flooding a network node with an endless stream of SYN packets, in some cases even when the port to which the SYN packets are addressed is a filtered port.

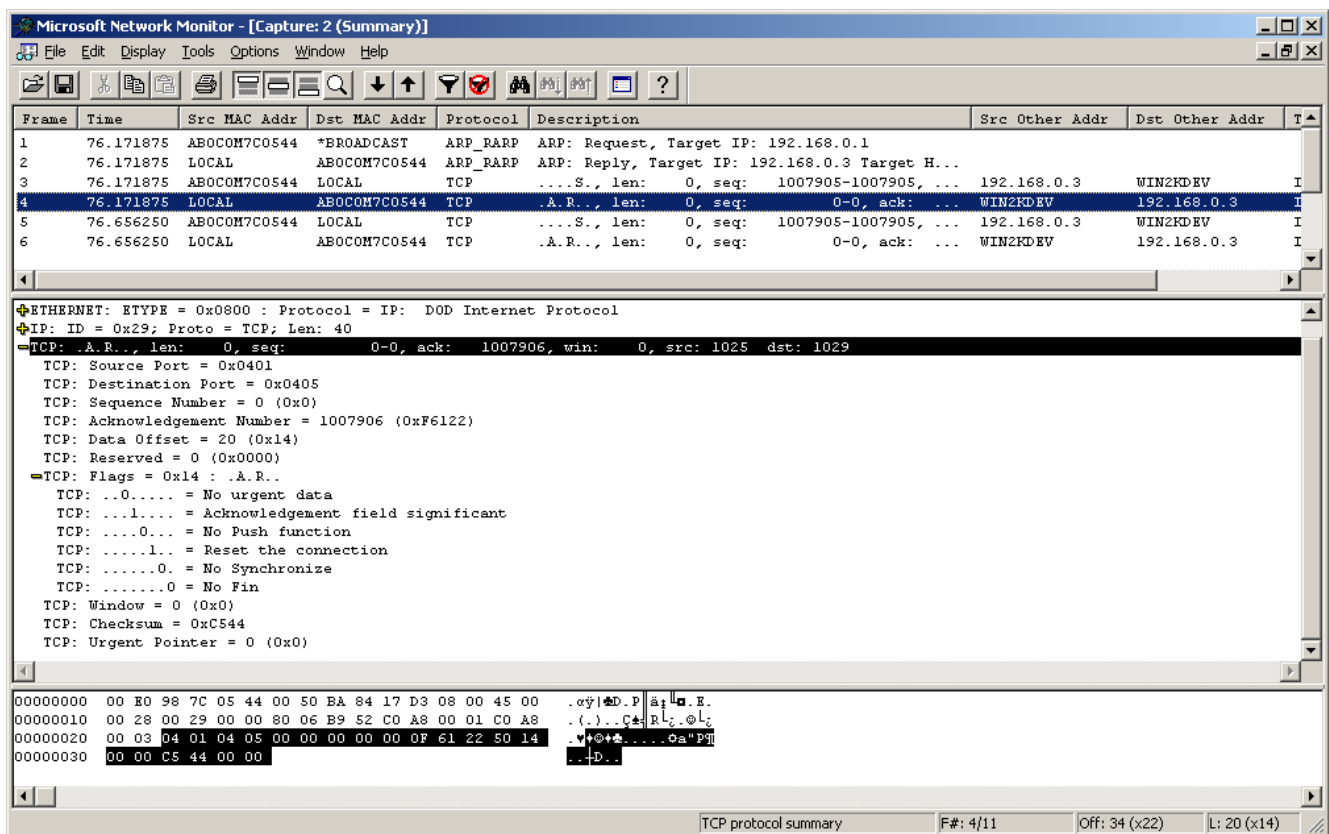


Figure 4-12: ACK Reset Packet Originating on Port 1025 to Shut Down TCP Connection Attempt

UDP packets to filtered ports often result in an ICMP message originating from the filtered port letting the sender know that the port could not be reached. When a network node tries to be helpful to remote nodes that attempt to connect to its local ports where services are not available the node is potentially wasting bandwidth and computing power just to be friendly. The unnecessary effort to be friendly can result in a DoS vulnerability or worse. There's no reason for a network node that does not provide services on a particular port to respond with an ACK/RESET or an ICMP packet letting the sender know of a failure condition. When TCP packets are unable to reach a network node's interface in the first place, the ACK/RESET packets are never transmitted and an attacker who sends such packets for malicious reasons has no way to know whether they did anything at all. An attacker who uses a port scanner can easily determine whether a network node is receiving packets and if so which ports are open and which are filtered when that network node tries to be helpful and sends failure notices or ACK/RESET TCP packets. There is currently no way to turn off such helpful behavior in Windows. To disable this type of network friendliness you must deploy a packet filtering firewall.

RFC 1918 Route Hardening

Private address ranges are those set aside by RFC 1918 for use in private networks that do not route to and from the Internet except by way of Network Address Translation (NAT) routers or firewalls/proxies. The NAT device or proxy may be assigned a globally routable IP address that it uses on its external interface to communicate with the Internet on behalf of clients (and possibly servers) located on the internal network that uses private

addressing in compliance with RFC 1918. Table 4-2 lists the address ranges defined as private and set aside by RFC 1918 for use in networks that do not require Internet routable addressing. The table also lists, in the last row, the address range set aside for Automatic Private IP Addressing (APIPA) by IANA, the Internet Assigned Numbers Authority.

Table 4-2: RFC 1918 and APIPA private address ranges

Address Range	Subnet Mask
10.0.0.0 – 10.255.255.255	255.0.0.0
172.16.0.0 – 172.31.255.255	255.240.0.0
192.168.0.0 – 192.168.255.255	255.255.0.0
169.254.0.0 – 169.254.255.255	255.255.0.0

The 10.0.0.0 address range grants its user 24 bits of the 32-bit IP address space to use for identifying subnets and hosts. As shorthand this address block is sometimes referred to as 10/8, meaning the addresses start with the number 10, and the first 8 bits of the 32-bit IP address range are significant for identifying a distinct network. The 172.16.0.0 address block is referred to as 172.16/12, meaning that the block begins at 172.16.0.0 and uses the first 12 bits as the subnet mask. Likewise, 192.168.0.0 and 169.254.0.0 are referred to as 192.168/16 and 169.254/16 respectively. Combined, these four address ranges, if all are used to address a gigantic private network, provide 17,956,864 unique IP addresses. More than enough for most private networks. If you have 17,956,865 nodes and absolutely have to address them all uniquely on a single network segment then, well, you've got a big problem.

The reason it's important to understand and harden every potential use of private address ranges is that DNS servers and firewalls often fail to implement RFC 1918 correctly due to oversights in product design or inadequately-hardened security configurations. One of the explicit requirements of any network or software service that allows RFC 1918 private network addressing is that the device or software do everything in its power to prevent private addresses from leaking outside of the private network that uses them. Not only do these addresses have ambiguous meaning outside of the private network, they represent a distinct security threat both as leaked configuration data that can be used by an attacker located outside the private network and as bad data that should be filtered out of things like DNS query lookup results that cross firewall or NAT private network boundaries.

RFC 1918 can be found on the Internet at <http://www.ietf.org/rfc/rfc1918.txt>

The worst case scenario of an RFC 1918-based addressing boundary violation is best exemplified by looking at real-world software bugs that exist in network software that lead to invalid security assumptions when private address information is entered purposefully into DNS by an attacker. One such software bug discovered by XWT Foundation and first fixed in Internet Explorer 6 service pack 1 pertains to the JavaScript same origin policy (SOP) which states that scripts and data are allowed to interact between frames in a frameset and between browser windows when the scripts and data appear to have originated from the same DNS domain. An attacker can therefore enter an RFC 1918 address into the authoritative servers for a domain that they control (either legitimately because the attacker registered the domain or by way of DNS hijacking) and cause a

Web browser to reference a local intranet network node using a DNS FQDN that shares the same DNS domain as another FQDN that references an Internet node controlled by the attacker. Because of JavaScript SOP and its DNS-based trust policy, the attacker is able to retrieve data into a browser frame or a new window from a server located behind the firewall on the private network and script the transmission of that data out to the attacker's Internet node by way of the FQDN with the matching DNS domain. The Internet node is trusted by JavaScript to be under the control of the same party who controls the server that provided the data contained in the additional frame or browser window based on the fact that they share the same DNS domain. The DNS resolver library in Windows does not currently permit you to configure a rejection policy for RFC 1918 addresses that are obviously bad because they do not reference a known local subnet or because they reference a known local subnet but they arrived in DNS lookup results for domains that are not authorized to use IP addresses in the local subnet. The only way to filter out such bad RFC 1918 addresses currently is through use of a firewall that supports this feature.

XWT Foundation's JavaScript SOP advisory URL is <http://www.xwt.org/sop.txt>

Another danger of bad RFC 1918 addressing information that crosses into a private network from an external source is the opposite of the JavaScript SOP bug discovered by XWT Foundation. Routing tables often treat RFC 1918 addresses as foreign and permit routing to and from these addresses across private network border routers. An attacker who controls a network node upstream from one they wish to attack could configure an RFC 1918 address on an interface in the upstream node and use it temporarily to attack the downstream node whose border router improperly relays traffic to and from the RFC 1918 address of the attacker's external node. When the attacker is done with the attack they can eliminate the RFC 1918 address binding on the interface in the upstream network node to make it more difficult to determine after the fact which upstream network node was responsible for the attack. If the attacker can physically attach and detach the attacking node to the upstream network, it's conceivable that the attack would become both completely untraceable when the attacker picks up the malicious equipment and carries it home and difficult to detect for an administrator who isn't watching for unauthorized RFC 1918 address usage within the upstream network. Whether or not you're worried about attacks that originate from external network nodes using RFC 1918 addresses, you should recognize that an attacker can easily cause certain information about network activity that originates from your private network to travel to their intelligence gathering boxes by taking advantage of routing tables that don't prevent packets with RFC 1918 destination addresses from leaving your private network. An RFC 1918 hardened routing table is one that explicitly lists each of the RFC 1918 address ranges and associates them with the loopback adapter interface rather than the real network.

Routing tables should be hardened with respect to RFC 1918 addressing in every device that is capable of originating network traffic, not just routers. There are a variety of ways that attackers can compel network nodes to send data to RFC 1918 addresses including the XWT Foundation JavaScript SOP bug example cited previously. A network node with an RFC 1918 hardened routing table will send data to itself through the loopback adapter rather than to other network nodes when attacked or hijacked using a bad addressing exploit. Preventing such a network node from sending data to private addresses that are

actually in use on the local subnet, however, is not prevented solely through hardening of the routing table. Use the following command to display the IP routing table in Windows:

NETSTAT -RN

The ROUTE command can also be used to view, add, change, and delete routes in the routing tables manually. The following command is synonymous with the NETSTAT command shown above, and Figure 4-13 shows its output on a box that has an RFC 1918-hardened routing table and the Microsoft Loopback Adapter:

ROUTE PRINT

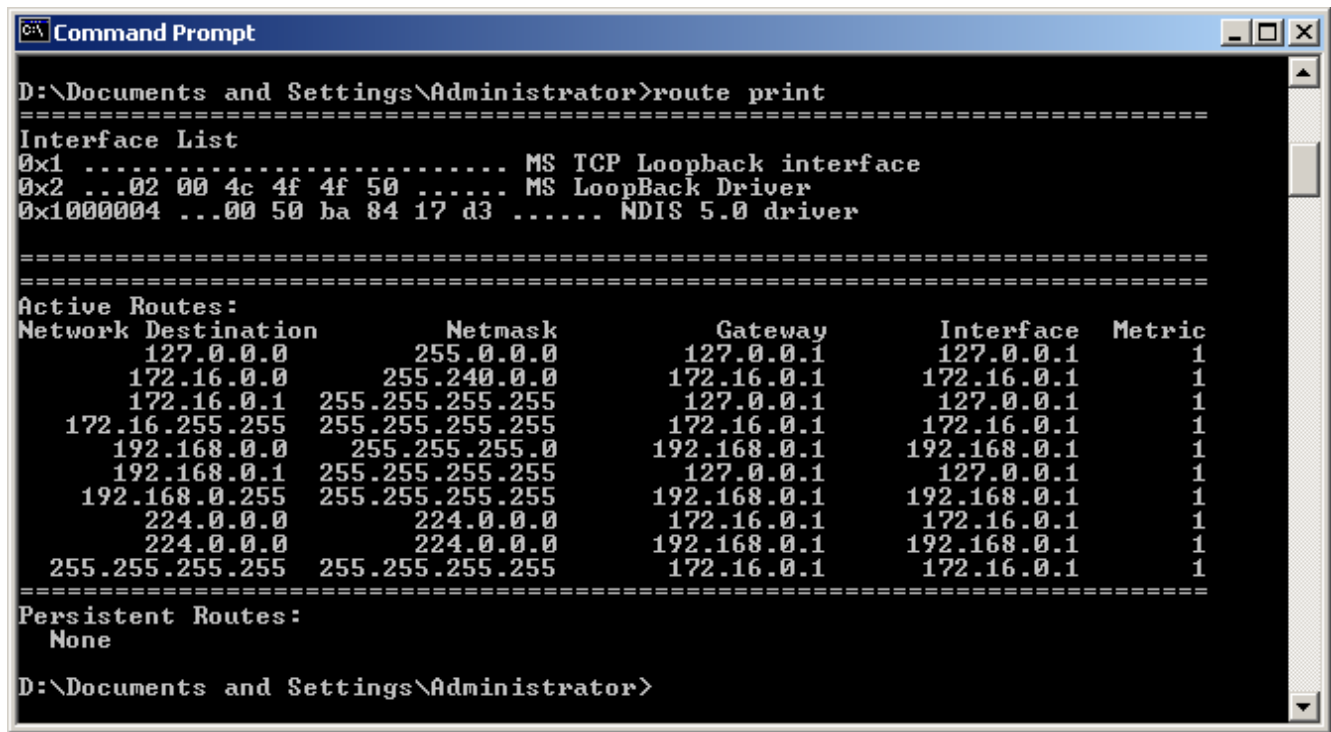


Figure 4-13: An RFC 1918 Hardened Routing Table Using The Microsoft Loopback Adapter

If a reverse proxy or similar port forwarding NAT router is used as a load balancing and security policy enforcement tool for your network, every authorized connection to your IIS box might come from the reverse proxy's IP address. If this is the case in your network, consider configuring the routing table on the box to communicate only with the single fixed IP address of the reverse proxy or NAT device. You may wish to select reverse proxy type security products based on the availability of this feature since the client IP address is mostly meaningless anyway except in certain scenarios. Such devices rewrite the source address of the forwarded packets so that server nodes receive and respond to them using a known local route rather than rewriting only the destination address in order to regenerate the packets on the LAN where a network node such as your IIS box will receive and respond to them using a potentially-malicious address that goes out on the default route. You might be concerned about losing client IP address information that is normally available to the server without a source address rewriting reverse proxy on the network, but for a variety of reasons, not the least of which is that source IP addresses

can easily be forged, the apparent client IP address is of little practical security value. You can easily live without it, and may even realize better security by doing so.

Windows Application Installation and Execution

Windows operating systems beginning with Windows 98 Second Edition make use of digital signatures to protect critical system files from tampering. In Windows 98 SE, digital signatures were limited to device drivers. As of Windows 2000 and beyond, just about any file can be protected and authenticated through the use of digital signatures. This brings up the obvious question, "what is the right digital signature for a particular file?" Because Windows must be able to validate a file's authenticity and trust automatically using software, Windows 2000/XP/.NET OS versions are configured to trust any code with any signature it believes to be valid. The problem is that you can't know at all times precisely the finite set of potential signers that Windows will trust. A bug in signature validation code, like the one publicized by thoughtcrime.org in August, 2002 where SSL certificate chain validation under Windows improperly allowed Internet Explorer to trust any certificate signed by any other authentic certificate issued and signed by a trusted Root certificate authority, could exist in Windows' signature trust validation code that would cause Windows to trust any signed code not just code signed by Microsoft. When a digital signature validates based on an unexpected chain of trust that relies on an unusual root certificate or a certificate issued in error by a trusted CA, there is no defense possible except human intervention to override the default behavior of Windows which is to accept as valid a digital signature that appears to be valid based on an automatic analysis of the signature by software. This default behavior is dangerous because software is always subject to bugs and the effects of other malicious software.

Since there is normally no user intervention allowed during validation of a digital signature, and therefore no opportunity for a human to inspect the chain of trust being relied upon by software, a malicious but verifiable digital signature is a very serious threat.

For more information about the SSL certificate chain validation bug in Windows see BugTraq archive <http://online.securityfocus.com/archive/1/286290> as well as <http://online.securityfocus.com/archive/1/273101> and IE 6 Service Pack 1 readme

Digital signatures are important for controlling code execution and thus they are important for information security. However, when used in automated systems they tend to authorize a broad range of code based on something that is, at its core, self-referencing. A signature is nothing more than a hash code encrypted with a private key. If you have the corresponding public key and believe you can trust it to actually correspond to the authentic private key that belongs to a trusted entity then you can use the public key to decrypt the hash code. When the decrypted hash and the dynamically-generated hash match, you can be sure that the item matches that which was digitally signed using the private key that apparently belongs to an entity you trust. You can also choose to trust that the entity you believe owns and exclusively controls the private key used to encrypt the hash when applying the digital signature was the entity that actually signed the data. But you don't know this for sure; you make presumptions of trust based on feelings of comfort and risk management policy.

The appearance of trust is based on a certificate with a particular chain of trust. Unfortunately, code that validates signatures rarely requires a specific, fixed, finite chain of trust; instead, the trust chain is discovered at run-time based on root certificates that are read into memory from a storage device. This makes digital signature trust chains extensible and configurable but it also makes them vulnerable to all sorts of real-world attacks and the unintended consequences of software bugs. Further, you can't be sure that a private key has not been stolen or compromised through cryptanalysis. Thus you can't be sure that a signature is authentic. Even if it does validate and even if the validation is based on a static chain of trust rather than a dynamic one, you still can't be absolutely certain based on signature validation alone that digitally signed code matches the code you wanted to trust or decided previously that you would trust.

Windows File Protection

Whether digital signatures are used or not, you still need to explicitly authorize code based on what it actually contains not based on who signed it. You don't want your computer executing code that is different today than it was yesterday. You know what the code was yesterday, so why not use that knowledge to protect yourself? You get that from hashes. Windows provides a hashing mechanism in conjunction with digitally signed catalog files that contain authentic hashes in order to implement Windows File Protection (WFP) and validate "digital signatures" of OS binaries and third-party vendor code certified by Microsoft as Windows Compatible. The WFP "digital signatures" aren't signatures applied to files but rather they are authentic hashes stored inside a digitally signed catalog. The fact that these catalog files exist and contain the list of authentic hashes that Windows trusts gives your Windows OS a substantial tamper-resistance. However, to circumvent WFP one need only replace a digitally signed catalog file containing authentic hashes with a digitally signed catalog file containing malicious hashes. The signature makes such an attack more difficult, but how much more difficult depends on whether or not there are bugs in its implementation (yes, there are bugs) and whether or not anyone malicious ever finds them (yes, malicious attackers will find them) and whether or not trusted signers are able to protect their secret keys from theft with absolute certainty. Absolute certainty is impossible. Therefore the signature isn't as crucial as the authentic hash.

A WFP mechanism that reads authentic hashes from a read-only storage or better yet a custom build of Windows source code that embeds hash validation and hard-codes the authentic hashes is superior to digital signature validation but it would create difficult technical challenges that nobody has tried to solve yet. WFP doesn't support such features today, but you should be glad it exists and uses digital signatures because a digitally signed list of authentic hashes of trusted binaries is a lot better than nothing. But that doesn't make digital signatures as a means to identify any and all trusted code preferable to validating authentic hashes as a means to prevent the execution of malicious code. To achieve that objective you have to stop all code from executing unless you have first authorized it based on its hash code. You then have to build a mechanism to protect the contents of and the access points to authentic hash storage. Digital signatures are one way to accomplish this, but ideally authentic hash storage would be done in hardware.

There is little doubt that at some point this will be commonplace in computers of the future. Chances are that digital signatures will take a back seat to authentic hashes, because the

former will fail in a variety of circumstances including the replacement of an authentic Root CA certificate with a malicious one that looks valid, whereas the latter will fail only in the circumstance of replacement of authentic hashes with malicious hashes.

Assuming, of course, that you're able to prevent malicious code from executing that would contaminate the automatic hash verification process. But since that's the point of the defense of either digital signatures or authentic hashes distributed along with executable code, you have to decide where you're going to pin your hopes. Pin them on signatures and trust anything that looks signed or pin them on hashes and have the absolute final word on what gets executed.

When you want to authorize a computer to execute different code, just update the authentic hash of the code that you choose to authorize. There's nothing wrong with relying on a digital signature to validate a particular hash as authentic, but it's unnecessary overhead at run-time because you already know the hash and you trust the hash (perhaps based upon previous confirmation of a digital signature that proved the hash to be authentic) and hashing algorithms are much more efficient than is asymmetric key decryption so dynamic hash verification using authentic hashes for comparison adds less processing overhead and can therefore be accomplished each time code is executed. The deployment to a Windows server of authentic hashes is currently accomplished via digitally signed catalog files in Windows File Protection.

Windows Update is compatible with WFP in that it delivers digitally signed catalog files containing authentic hashes along with code. The hashes are trusted as authentic only after being verified by way of a digital signature applied to the hashes. Provided that the code's hash matches the authentic hash contained in the digitally signed catalog file, Windows Update will trust the code and install it automatically. It also places the signed catalog file on the hard disk so that it can be used again later to verify the authenticity and trust of installed OS code and third-party code such as device drivers. A related service for automatic code installation, Software Update Services, provides a way for companies to host their own Windows Update server for approved code updates and hot fixes to propagate code and digitally signed hash catalogs securely and automatically to a network of managed Windows boxes.

Windows stores signed hash catalog (.CAT) files in the System32\CatRoot directory. As you can see in Figure 4-14, a catalog file contains a Tag identifier label for each of the binary files whose hashes are contained within the catalog. For each Tag, the security catalog stores the hash (called a Thumbprint in the figure) and the hash algorithm used to create the hash. This information permits Windows File Protection to reproduce hashes using the right algorithms during file hash verification. You can simply double-click on a .CAT file to open the Security Catalog window.

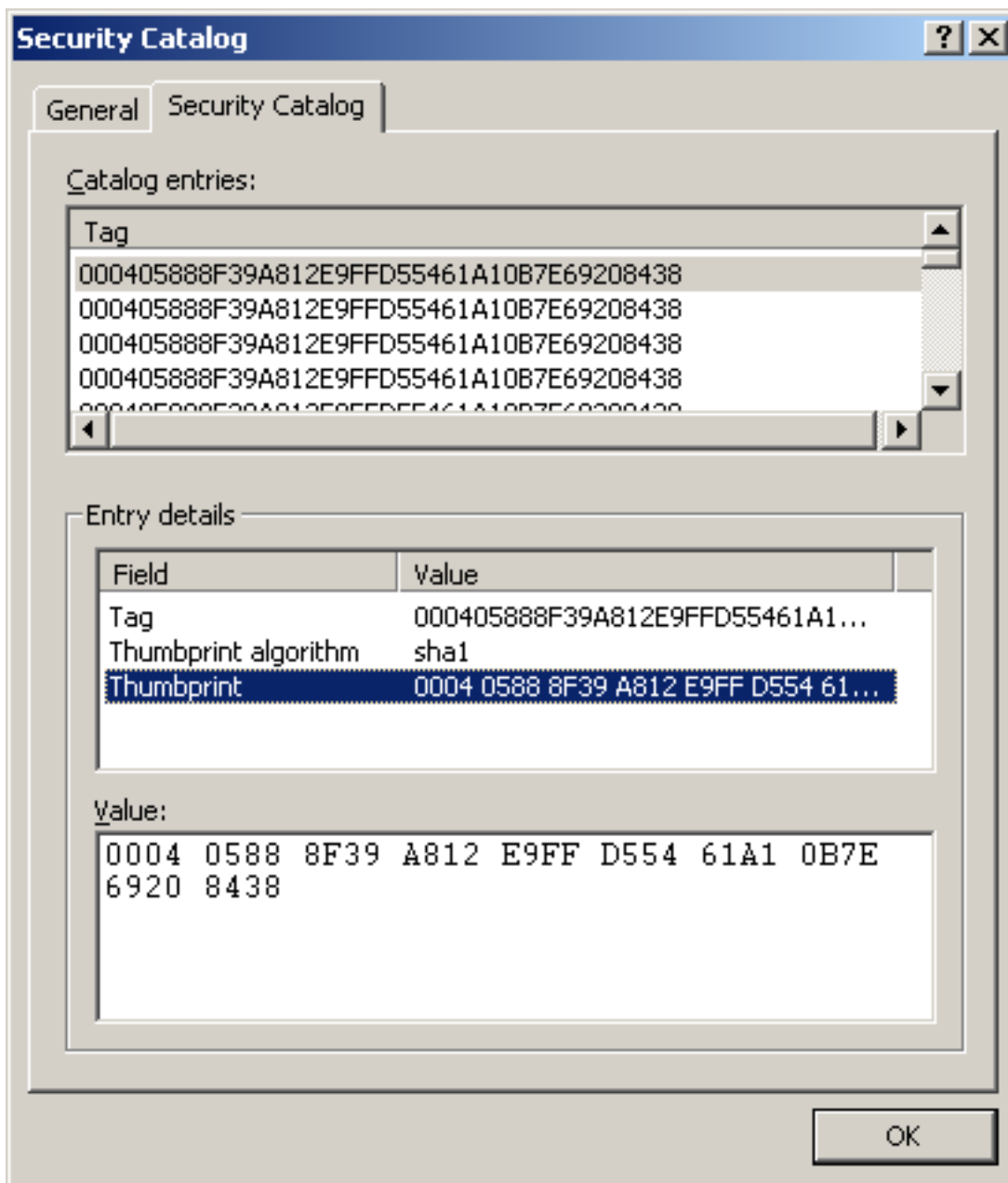


Figure 4-14: Double-click on Any Digitally Signed .CAT File to View The Hashes It Contains

Copies of protected files with authentic hashes stored in security catalog files are automatically placed in the System32\dllcache directory by Windows. In addition, WFP knows where the original authentic binary came from, its source media, based on which signed catalog file it finds the authentic hash inside. Windows keeps track of the source of each signed catalog file so that it can automatically restore files from the right place or display a prompt requesting the correct source media be inserted. This makes it possible for WFP to replace a service pack or hotfix binary with the binary from that same service pack or hotfix distribution rather than making the mistake of restoring an older version when a newer one should be installed instead. However, WFP is dependent upon the existence of authentic hash catalog files within System32\CatRoot for this logic to function properly. A copy of certain authentic catalog files, such as NT5.CAT and NT5INF.CAT, are placed in dllcache as well because these files are also protected

system files. This prevents the authentic hashes of the most important system files provided as the core of the OS installation from being deleted by an attacker. However, service pack and hotfix catalog files that contain updated versions of protected files whose authentic hashes are contained within these protected catalog files are not protected files, which makes it possible to compel WFP to rollback to pre-service pack or pre-hotfix versions if an attacker is able to delete the corresponding catalog files installed by these updates.

A quota can be set on the use of dllcache by Windows to prevent it from using more than a predefined amount of disk space. See Knowledge Base Article Q222473 entitled "Registry Settings for Windows File Protection."

When WFP detects a change to a protected binary that causes the protected binary to fail the authentic hash verification, WFP replaces the untrusted binary with the trusted one first by looking for an authentic copy stored in dllcache. WFP checks the hash of the copy stored in dllcache, but it does so in a rather strange way. Instead of looking for the file in the Security Catalogs stored under System32\CatRoot by its filename, WFP looks by its Tag value. The Tag value is not linked to filename, and it is therefore possible to place an authentic copy of a different protected binary in the dllcache directory and give the copy the wrong filename on purpose. Doing so causes WFP to authenticate the binary based on the fact that it has a valid hash as determined by its Tag record in a security catalog file, but WFP then assumes the file is the right file to restore under its apparent filename and does so. This trivial attack method completely fools WFP and creates a DoS condition due to the fact that the wrong code is loaded from the file which in spite of its filename contains a copy of some other protected binary. Since the OS doesn't use the WFP Tag value when loading binary modules into memory to execute them, there is a simple disconnect here between the way that WFP locates the right authentic hash to use from the signed catalog files and the way that binary modules are actually located and used by the OS. In addition, the same odd behavior is exhibited in the way that authentic signed hash catalog files are protected such as the NT5.CAT file. You can place an authentic copy of a different security catalog file in System32\CatRoot and give it the filename NT5.CAT and in so doing cause WFP to believe that the authentic NT5.CAT file is still present in the CatRoot directory. What this means is that all of the authentic hashes certified by the real digitally signed security catalog file cease to be protected files and Windows File Protection stops paying attention to them. Keep a tight control on your CatRoot directory, and use the Windows File Checker and Signature Verification utilities periodically, as described in the next section, because these tools will warn you when protected binaries no longer appear anywhere in any security catalog, based on Tag value. When this condition arises, there is no automatic notification provided at run-time, authentic binaries that once were protected become vulnerable to tampering as WFP no longer considers them to be protected.

Windows File Checker and Signature Verification Utilities

Windows File Protection comes with two utilities that enable you to actively manage the hash verification process and obtain WFP scan reports that help you understand the contents of digitally signed .CAT files stored under System32\CatRoot where authentic hashes used to verify protected files are stored. When security catalog files are missing or damaged through malicious acts, software bugs, or administrative mistakes these utilities are often the only way to discover that WFP no longer considers certain files to be

protected. The first utility, known as the Windows File Checker, is accessed through the SFC.EXE command-line program. Use this program to request an immediate scan of all protected files, control settings for automatic scan at boot time, and manage the System32\dlldata directory. The command syntax is shown below along with its available parameters.

```
SFC [/SCANNOW] [/SCANONCE] [/SCANBOOT] [/CANCEL] [/ENABLE] [/PURGECACHE]
  [/CACHE SIZE=x] [/QUIET]
```

Both /PURGECACHE and /CACHE SIZE=x refer to the System32\dlldata directory, the location of the WFP authentic file cache. To request a scan of all protected system files one time at the next system boot, specify the /SCANONCE parameter. An immediate scan is requested using /SCANNOW and the existing contents of dlldata or /PURGECACHE and an empty dlldata. To schedule repeated scans, use /SCANBOOT to set the Registry key value for a full scan every time the system boots. During a scan, files protected by WFP for which authentic hashes are available inside security catalog files under System32\CatRoot are hashed and compared against the authentic hash stored in the authentic hash catalog (.CAT) file, if any. The SFC command-line utility reports file replacement actions as well as files with missing or invalid hashes that couldn't be verified, plus failed attempts to restore valid files, by way of the System Event log.

For a protected file scan using WFP authentic hashes and digitally signed catalog files that results in a protected file verification report stored as a text file that is easier to read and doesn't clutter up the Event log, a signature verification program (SIGVERIF.EXE) exists. Figure 4-15 shows the file signature verification user interface, a simple dialog-based Windows application. Follow the prompts to select a storage location and search parameters, including whether or not you'd like a complete list of every file that does not contain a signed hash within one of the security catalog files, and SIGVERIF.EXE does the rest. By default the program stores a text file output report in the system root directory named SIGVERIF.TXT.

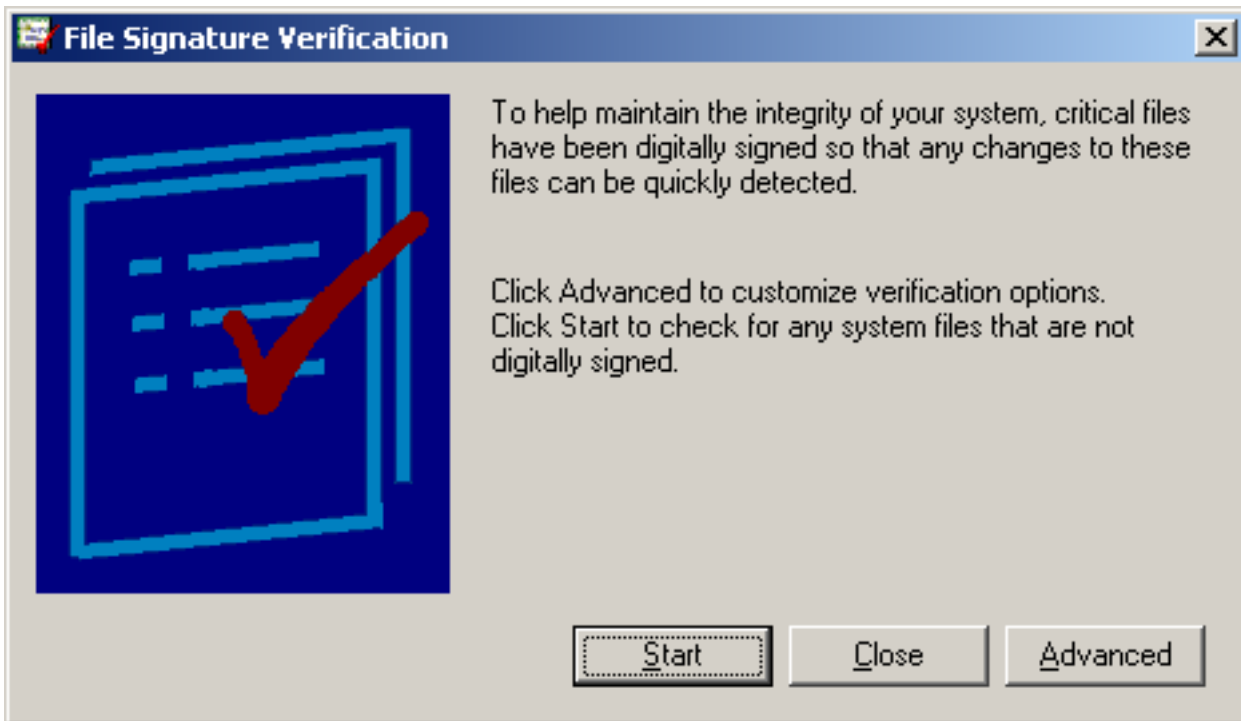


Figure 4-15: The SIGVERIF.EXE Windows File Protection File Signature Verification Program

It's important to note that the Windows loader does not verify digital signatures each time a compiled code module is loaded into memory for execution. This means that any code that is able to inject itself into the process space created by the OS, and any mechanism that an attacker could use to alter what the loader puts into memory as it tries to read bytes from the file system, are still vulnerabilities in the platform in spite of the introduction of digitally signed catalog files that contain verifiable hashes of OS and third-party binary modules. Further, tampering in-memory is still possible because Windows doesn't verify authentic hashes at all during program execution. Finally, because the "digital signature" that is applied by Microsoft to Windows protected files is actually stored in a security catalog file rather than within the files themselves, you have yet another sensitive data storage location under System32 that must be protected at all costs.

The value provided by these security catalog files is directly dependent upon careful monitoring to make sure they exist as expected and still contain authentic hashes for every file that is supposed to be protected automatically by WFP, as these security catalog files are the new keys to the kingdom; a little better than nothing, which was the protection we had in the past, but still not foolproof.

The Windows platform was, at first, a fish out of water on the Internet. TCP/IP was an afterthought for Windows, and for many years there were only closed, proprietary, incompatible programming interfaces for TCP/IP software under Windows. Eventually, Windows Sockets (WinSock) changed that situation, providing an open, standardized TCP/IP programming API for Windows based on Berkeley Software Distribution's Unix (BSD) socket programming API. And when Microsoft provided native support for TCP/IP in Windows to go along with the WinSock API, everything started to get TCP/IP-enabled. Well, this chapter showed you a little of the harsh reality that exists now as a result.

TCP/IP is everywhere in Windows, even in places you don't want it to be, and you can't get rid of it. The best that you can do is learn to contain it and achieve a relative hardening of your IIS box compared to doing nothing.

Windows is still a fish out of water with respect to TCP/IP and the Internet. Its security primitives appear on the surface to be time-tested and hardened-by-fire (not to mention pain) having overcome severe deficiencies in legacy code to provide a feature-rich foundation for secure network computing. But in reality there will always be too much complexity in Windows, as a direct result of its programmers' love of new features and its pervasive developer-orientation. But with digital signatures providing a measure of trust to certify authentic hashes of legitimate operating system code modules, and with security-aware application development tools and techniques, there is little doubt that there is potential for productive, safe use of the Windows platform as the basis of Internet (and TCP/IP intranet) servers. The real question you have to answer for yourself is whether the risk of a platform you know to be flawed is worth the reward of being able to take a few shortcuts and ignore everything that goes on behind the scenes so that you can focus on where you want to go today. In my opinion, based on all that I have seen both publicly and privately at Microsoft, and based on my own analysis of vulnerabilities in Windows, Microsoft cannot be trusted to build a "behind the scenes" that is reliable and trustworthy. Either they release the source code to Windows and each of the other Microsoft products or I personally have decided to stop allowing Microsoft code to execute on my microprocessors.

Chapter 5: ASP.NET Security Architecture

Security in ASP.NET is tightly coupled with security for Internet Information Services due to the fact that IIS provide the host environment for ASP.NET run-time functionality. The only completely secure platform for hosting ASP.NET applications is provided by Windows .NET Server and IIS 6. Under IIS 6 there is no such thing as in-process application code; all application code executes in worker processes (instances of w3wp.exe) in the security context of Network Service, a built-in account with very few privileges on the operating system. This architectural change confines the potential damage that malicious code can do when it is published to the server. ASP.NET provides a surrogate process model configuration option for use under IIS 5, but the protection it provides is superficial and limited only to protecting against attacks launched on your server through the use of ASP.NET code which is more secure to begin with and therefore less of a threat so IIS 5 remains relatively unsecured in many deployments. To host secure ASP.NET applications under IIS 5 requires substantial security preparations to be performed in advance by a system administrator who understands IIS security.

Whether you deploy ASP.NET applications under IIS 6 or a security hardened IIS 5 and Windows 2000 server, the security facilities provided by ASP.NET allow you to build password protection, encryption, and code access and role based security policy into Web applications. Protection from malicious code is automatic in the .NET Framework at the level of memory heap and stack protection due to Common Language Runtime memory management services. As a programmer your job is to ensure that only authorized users and certified code are allowed to perform operations within your ASP.NET applications. As an administrator your job is to configure ASP.NET security settings so that code executing within the ASP.NET script engine is incapable of doing harm to the server and restrict developers' access to system resources according to security policy. This chapter helps you accomplish these tasks.

There is nothing more important to consider as you write code or manage its deployment than the question: "How do I know that the caller, the program or user that invoked the code, is authorized to carry out the operations made possible by the code?" However, the reality for programmers who built applications in the past with just about any programming language and development environment was that very little could be done technically to impose additional security policy decisions on top of any application-wide policy settings supplied by the external security features of the operating system and runtime host architectures. As a result, developers would consider this question and determine that the only practical answer was to trust the caller and hope for the best, leaving security as an administrative worry.

Built-in Security Primitives

Developers tend to passively assume that any caller that is able to satisfy (or bypass) the security restrictions implemented by the OS and application host environment must be an authorized, authentic caller. Any password protection or other credential-, identity-based access restrictions needed by an application to protect against unauthorized access tend to exist only at boundaries between servers or services due to limitations of computer

security, historically. ASP.NET provides Web applications built around IIS with full access to the enhanced security features of the .NET Framework in addition to implementing Internet standards for password protection. Every line of managed code hosted by the .NET CLR can define specific security requirements, giving the ability to demand that all callers, both direct and indirect, possess certain permissions. This .NET feature, known as Code Access Security, enables Role-Based Security which reviews the permissions granted to a particular user based on the group or groups the user belongs to. Role-Based Code Access Security adds a sixth facet to security in client/server network software which is logically divided into the following parts:

- Authentication (credentials)
- Authorization (permissions)
- Session Management (state tracking)
- Impersonation (server's effective security context)
- Encryption (data protection and digital signatures)
- Role-Based Code Access Security (system resource protection)

Typical Web applications implement authentication and authorization, which together make password protection possible, in combination with automatic session management to enable users to use a Web site anonymously until they need to complete a task that must be associated with a particular user identity at which time they authenticate to receive authorization to conduct the task in the name of the authenticated identity. For many Web applications, authenticated sessions enable persistence of session state information for a particular user without the concerns normally associated with computer security. Any malicious user who manages to break into a user's authenticated session state is able to see what that user has been working on while using the site such as which items the user has added to a shopping cart for future purchase or what the user has searched for previously, whatever the persistent user-oriented session state happens to contain.

Sessions, even authenticated ones, are not secure: they can be hijacked even if encryption is used to protect them. There's no reason for many sites to worry about this fact, however, because the damage done by the hijacker is either trivial, such as changing the contents of another user's shopping cart, or zero, no impact whatsoever and no breach of confidentiality.

To properly secure client access to ASP.NET application code you must understand the difference between authenticated sessions and secure authenticated sessions that include cryptographically valid credential verification that is trustworthy enough to use in place of authentication to allow impersonation on the server. Impersonation is the key to server security because it establishes the limitations imposed by the server on any unauthorized actions that malicious users or malicious code can potentially unleash. ASP.NET provides several options for easily and correctly implementing cryptographically valid secure authenticated sessions.

Authentication

Users prove their identities to the satisfaction of a network server by supplying credentials that are validated through a process known as authentication. Credentials, and the mechanism by which they are transmitted for authentication in a security system, are the

points of highest vulnerability for network security. A user ID and password are the most common technique used for managing and transmitting network credentials. User IDs are often relatively easy to remember and some systems use ID naming conventions that are nearly impossible for users to forget such as e-mail address or first and last name. Users appreciate IDs that are easy to remember and often select passwords that are also easy to remember. A system that is protected by an e-mail address and an easy-to-remember password has little real protection regardless of the other security mechanisms employed because any person who knows the e-mail address of the user and can guess or crack the password will be able to break in successfully. Cryptography helps protect against eavesdropping by hackers but anyone can establish an encrypted connection with a network server if they have physical access to the network, so encryption should not be thought of as a locked door through which unwanted visitors are unable to pass.

Authentication can be a processing bottleneck unless shortcuts are taken to improve performance of network servers that restrict access to only authenticated users. However, the wrong shortcuts will render a network completely insecure in spite of the best technology and security policy in other respects. To improve performance and enable a single server to service many more simultaneous users, some sites opt not to use any form of encryption to protect the transmission of user credentials. Or worse, some sites integrate session management with authentication in a way that is not technically valid such as by dropping a cookie that contains a unique session identifier and then allowing the user to authenticate after which the site marks the session as authenticated. Such sites commonly disregard the obvious danger signs of attempts by hackers to discover authenticated session identifiers including repeated requests coming in with invalid session identifiers or requests with valid session identifiers originating from two or more IP addresses simultaneously. It may seem like common sense that a single authenticated user can't be in two places at once, but building common sense into network software is easier said than done.

Authorization

Authorization is the process by which an authenticated identity's access permissions are evaluated according to an appropriate security policy and a decision is made to allow or deny a requested operation. Authorization may occur explicitly through the use of explicit permissions settings for users and groups or through custom authorization logic. Authorization always occurs implicitly through the use of operating system access control, user security context, and automatic .NET Framework security features.

Session Management

Sessions are tracked and managed by server software with respect to clients in either a stateful or a stateless fashion depending upon the network application protocol design. Many network services begin and end a logical session with every request. For example, DNS queries return fully qualified domain name or IP address lookup results to the client and immediately wrap up any server-side processing for the client's request, releasing resources allocated during processing and effectively forgetting that the client had ever made its request. No state is preserved on the server to allow the client to make

subsequent requests and associate them logically with previous ones so the protocol is said to be stateless.

Changing from stateless to stateful after the fact is not as simple as you might think because backwards compatibility must be maintained and client software sometimes depends on the server to close the connection in a connection-oriented stateless protocol like HTTP in order to indicate the end of the server transmission. For DNS as it exists today there would be no practical benefit to switching to a stateful protocol since DNS clients are not required to authenticate with DNS servers and DNS doesn't meter or otherwise control access to DNS lookup services. FTP servers, however, do need to maintain session state while communicating with clients because logins are required, file transfer preferences must be tracked for the user's session, and FTP servers must track data such as an FTP user's current working directory in order to process client requests for file transfer operations and enable remote navigation of the server's filesystem hierarchy. By contrast, HTTP is a stateless protocol designed to allow for some level of security and a large degree of extensibility. As a network application protocol HTTP is closer to DNS than to FTP, and any state management required by an application built around HTTP must be built by the developer as part of the application code. HTTP servers don't need to track sessions for users in default configurations because like DNS servers HTTP servers simply receive requests and send responses then immediately clean up after themselves and forget about the request just processed. Any session tracking must be built on to the base protocol as custom application logic. Web browsing lends itself to application-specific session tracking in spite of its stateless design due to multiple mechanisms available in the HTTP and HTML specifications to retransmit server-defined data with each subsequent request as the user navigates from page to page and from site to site.

Impersonation

Impersonation is the setting on a per-request basis of the Windows account user security context by which a process or thread identifies itself to the operating system.

Impersonation is based either on verification of the credentials provided by the user or the configuration settings for the application. Network servers that allow guest users who don't have credentials support a type of impersonation called anonymous. If anonymous requests are allowed then anonymous impersonation occurs in order to ensure that application code executing on behalf of anonymous users is afforded only restricted rights in an appropriate Windows security context. A token that represents the restricted anonymous impersonation account is passed to the code that protects resources requested by the anonymous user. When access is limited to authenticated users only, application-specific code can dynamically set the impersonation context for the request. A special feature of ASP.NET enables automatic impersonation where any authenticated identity is mapped to its corresponding Windows user account and that account's security context is used as the impersonation context for the request.

Encryption

Encryption is an aspect of cryptography that pertains to the transformation of data from an original form referred to as plain text into a protected form, referred to as cipher text,

through application of an encryption algorithm, referred to as a cipher, and a key that controls the way in which the cipher is applied to transform plain text data. There are two fundamentally different types of cipher. The first, called symmetric, relies on a shared secret key to both transform plain text into cipher text in an encryption operation and also transform cipher text back into plain text in a decryption operation. Since the same key is used for both sides of the cryptographic operation, protecting the secret key from interception by third-parties is more important and just as complicated as making the cipher computationally difficult to crack without access to the secret key.

The second type of cipher, called asymmetric, uses a pair of different but complementary keys for encryption and decryption transformation operations. Cipher text produced through application of one key can only be decrypted through application of the complementary key from the key pair. The benefit of this approach is that one of the keys from the key pair can be kept secret and the other given out publicly such that any third party can decrypt cipher text if they know the identity of the encrypting party. As long as decryption succeeds using the key given out publicly by the encrypting party, the recipient knows with some certainty that the cipher text came from the party who holds the complementary secret key. This is the basis of digital signatures known as public key cryptography. Digital signatures are able to replace the less-secure but more-common shared secret authentication through the use of certificates which are digitally signed credentials certified by a third-party.

In addition to serving as the basis of digital signatures, asymmetric ciphers enable encryption and decryption to be performed as well through the use of two complementary key pairs. Each side of an encrypted communication holds one key as a secret and exchanges with the other side the complementary key. Each side encrypts plain text using the public key provided by the other party and then sends the resulting cipher text which can be decrypted through application of the secret key held only by the recipient. The generation of keys pairs and the procedure for key exchange that must take place before two parties can exchange encrypted messages using an asymmetric cipher are additional complications that make this type of cipher more challenging to deploy. The added security provided is often worth the extra effort.

Code Access Security

All code executes in a particular user's security context determined by impersonation. The most common type of impersonation occurs when an operating system allows an interactive user with a conventional local user account to log in while physically sitting in front of the computer. The programs executed by the interactive user use that interactive user's security context. Services that run automatically when the operating system boots up can be set to impersonate, or execute on behalf of, a particular interactive user account security context. The operating system also provides special system security contexts that don't correspond to interactive user accounts and either type of security context can execute code.

The code that a particular security context is capable of executing depends first on the filesystem permissions that restrict access to the binary code stored in an executable file or DLL library and next on any application-specific security policy implemented by the shell, runtime, or host environment inside which the request to execute code originates.

The common language runtime implements a comprehensive security policy called Code Access Security for controlling access to system resources, files, managed and unmanaged code. Code Access Security is automatic in ASP.NET and you may not have to think about it beyond the level of role based security as described in this chapter. For details on developing custom class libraries that make use of Code Access Security see the .NET Framework documentation.

Internet Information Services ASP.NET Host

IIS host the ASP.NET Script Engine which runs as managed code inside the Microsoft .NET Framework on Windows .NET and Windows 2000 servers. Security configuration settings for the ASP.NET platform are your first line of defense against improper use of your code and the platform itself to compromise data integrity and privacy. These settings represent the passive security features of ASP.NET. For your applications to achieve complete security you must also take advantage of the active security features and encryption facilities discussed later in this chapter.

ASP.NET Script Engine

The ASP.NET script engine, `aspnet_isapi.dll`, is loaded by IIS to process files associated with ASP.NET applications. Configuration and source files are protected from access by clients through the ASP.NET script engine. Figure 5-1 shows the response produced when a browser attempts to access a protected ASP.NET file.

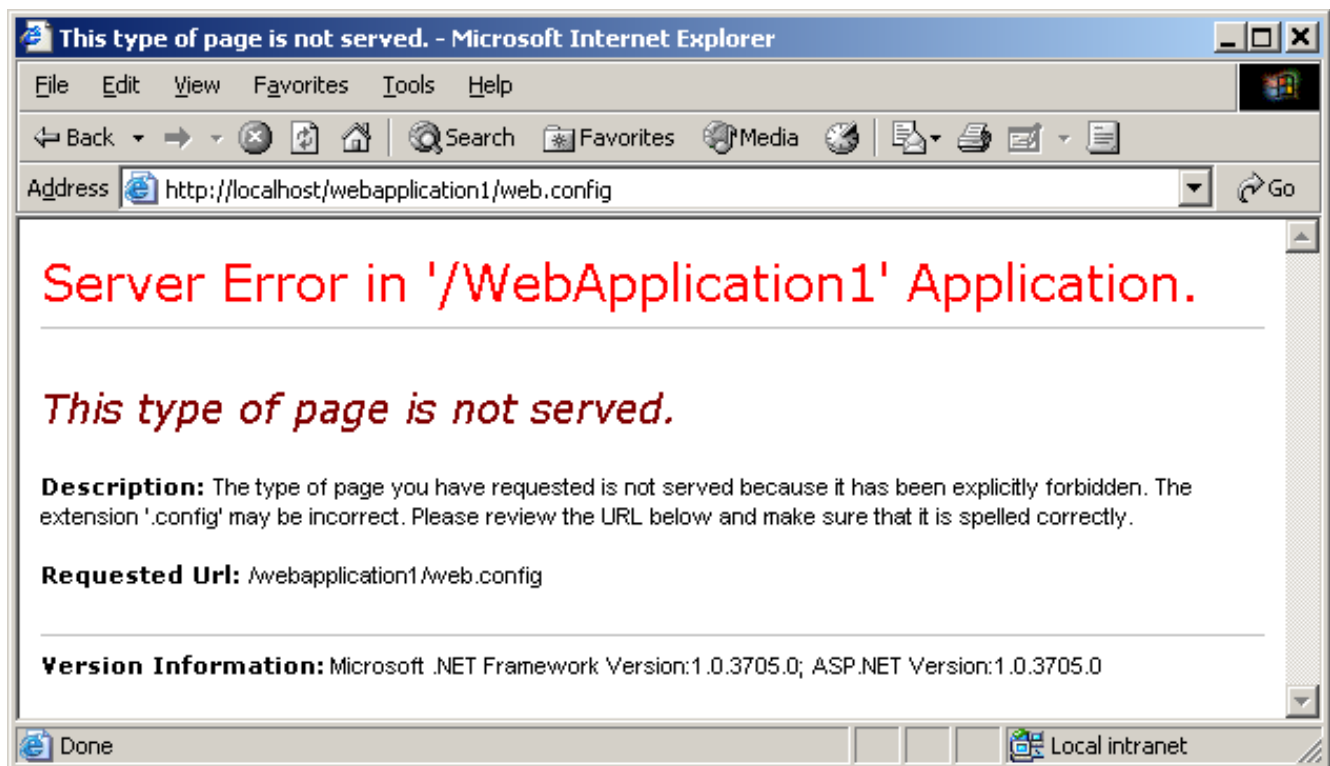


Figure 5-1: ASP.NET Configuration and Source Files Are Not Served

`Aspnet_isapi.dll` is an ISAPI extension DLL that is loaded by IIS in order to handle request processing for files of a particular type. IIS associates an ISAPI extension DLL with file

types based on a list of file extensions known within an IIS-hosted application as Application Mappings. Understanding the file types that IIS associates with ASP.NET script engine ISAPI extension DLL is the first step to understanding security in ASP.NET.

ASPNET_ISAPI.DLL File Types

ASP.NET security is only applicable to the file types for which the ASP.NET script engine ISAPI extension DLL, `aspnet_isapi.dll`, is configured in the Web application's App Mappings. As you can see in Figure 5-2, Application Mappings in each IIS Application specify the ISAPI extension DLL that IIS will use to service requests for specific file extensions. Additionally, the WWW Service Master Properties define the default Application Configuration for all Web sites hosted under IIS on a particular server box. To edit the WWW Service Master Properties you open the Properties window for the Web server inside MMC. To edit the Application Mappings for a particular IIS Application, you open the Properties window for the Application root directory. Any directory hosted by IIS can be marked as an Application root.

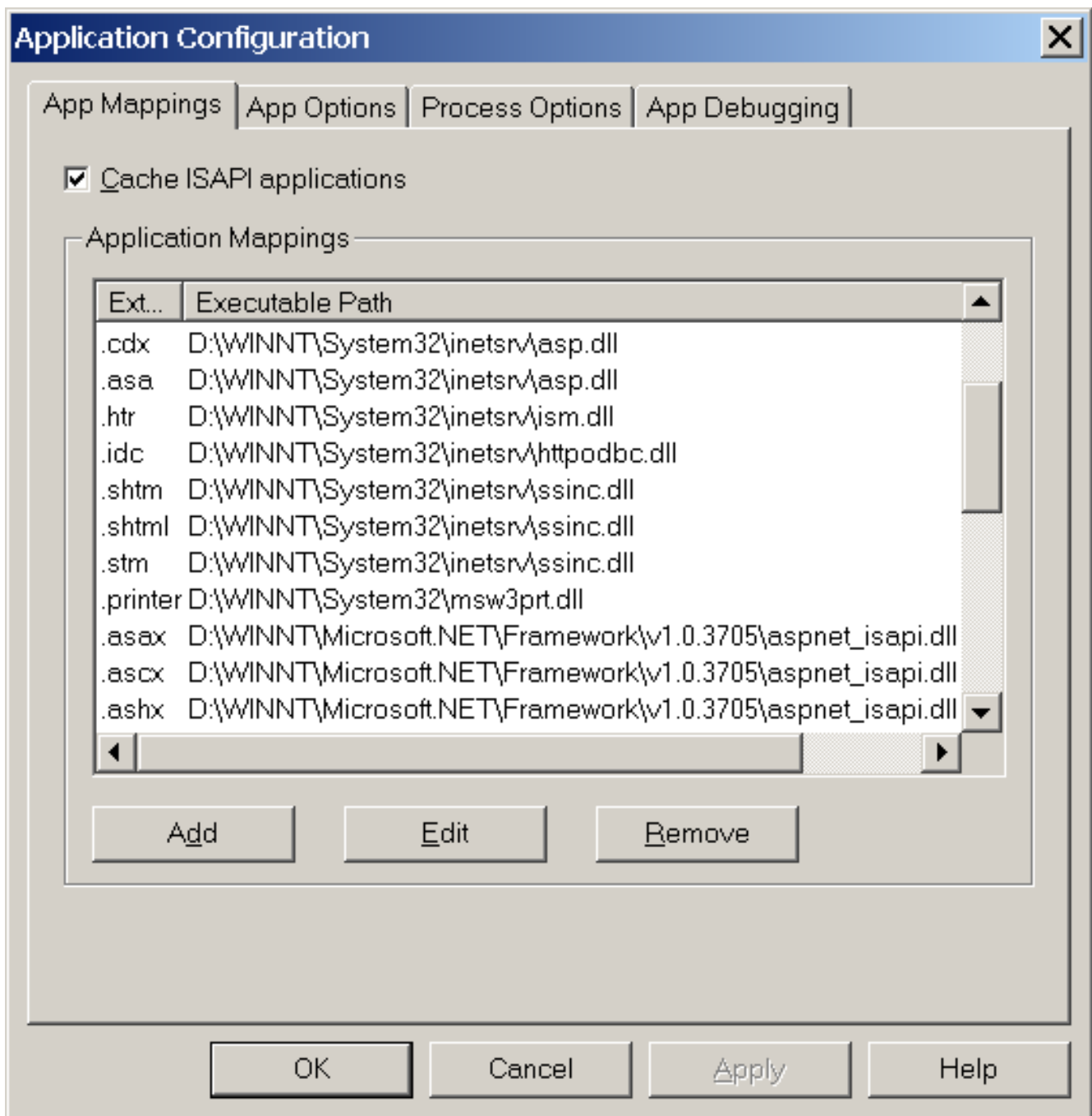


Figure 5-2: Application Configuration Settings Apply for Each IIS Application

All file types for which a different ISAPI extension DLL or script engine is configured, and all file types for which no special Application Mapping is established, will be processed within IIS by code that bypasses ASP.NET security settings. By default `aspnet_isapi.dll` is registered to handle request processing only for files that end with `.asax`, `.ascx`, `.ashx`, `.asmx`, `.aspx`, `.axd`, `.vsdisco`, `.rem`, `.soap`, `.config`, `.cs`, `.csproj`, `.vb`, `.vbproj`, `.webinfo`, `.licx`, `.resx`, or `.resources`. All other files types are handled by another script engine, a different ISAPI extension, or by a built-in feature of IIS. This includes all `.html`, `.htm`, `.asp`, `.jpg`, `.gif`, `.txt`, and everything else. To make sure that every request for every file in an ASP.NET application is processed with awareness of ASP.NET security settings you must add an extension to Application Mappings for each file type, including graphic file types, that

exist in your ASP.NET application directories. Click the Add button to bring up the window shown in Figure 5-3 then enter a file extension and browse for the full path of the aspnet_isapi.dll script engine.

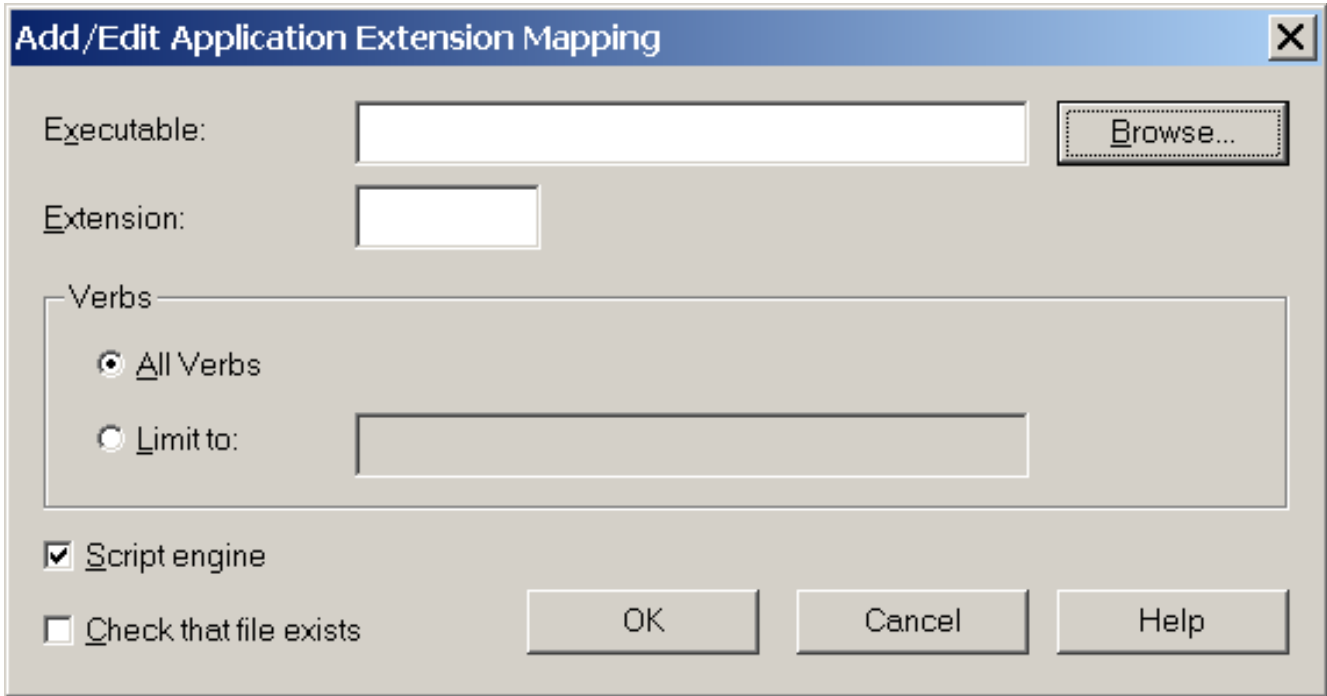


Figure 5-3: Add Application Extension Mappings for ASP.NET

Enforcing ASP.NET security for every request including requests for graphics adds processing overhead that may be substantial in your deployment. You may wish to use the default Application Mappings unless it is important to the security of your ASP.NET application to authenticate each request using aspnet_isapi.dll. For each file type that you map to aspnet_isapi.dll in IIS you may need to edit machine.config to add or edit the corresponding line in the list of httpHandlers. An excerpt from the default machine.config file is shown here:

```
<httpHandlers>
<add verb="*" path="*.aspx" type="System.Web.UI.PageHandlerFactory"/>
<add verb="*" path="*.config" type="System.Web.HttpForbiddenHandler"/>
<add verb="*" path="*.asp" type="System.Web.HttpForbiddenHandler"/>
<add verb="GET,HEAD" path="*" type="System.Web.StaticFileHandler"/>
<add verb="*" path="*" type="System.Web.HttpMethodNotAllowedHandler"/>
</httpHandlers>
```

As you can see, machine.config maps file types to handler classes. By default System.Web.HttpForbiddenHandler is mapped to .asp and .config files and GET and HEAD methods are allowed in HTTP requests for any static (non-script, non-program) item that doesn't match an explicit mapping.

Windows .NET/2000 Server OS or Windows XP Professional

ASP.NET only works under Windows 2000 Server, Windows .NET Server or Windows XP Professional because the security, performance, and architectural improvements of IIS 5 on Windows 2000 or Windows XP Professional and IIS 6 on Windows .NET Server over IIS 4 on Windows NT Server are profound. Attempting to shoehorn ASP.NET's secure server platform into IIS 4 with its flat process model and Microsoft Transaction Server (MTS) integration for process boundary protection seems pointless when you consider the benefits provided by Windows 2000/XP and .NET Server. Instead of relying on MTS for hosting out of process Web applications through instances of mt.exe, IIS 5 and Windows 2000 incorporate native support for COM+ services that enhanced both COM and MTS, combining the best aspects of each and giving them a new name.

Implementing ASP.NET around COM+ and IIS 5 provided the optimal starting point and it's therefore unlikely that ASP.NET will be released in the future for IIS 4.

Windows operating systems on which ASP.NET can be run provide improved security mechanisms for some of the fundamental requirements of Web applications. An important one is enhanced support for maintaining impersonated identities across process boundaries. The original COM, predecessor to COM+, was not designed to preserve the per-thread security context and token established for a particular thread as it carries out request processing on behalf of an impersonated identity. Instead, COM would revert to the security context and token of the process that hosts the thread (under IIS 4 that would typically be SYSTEM for in-process applications or IWAM_MachineName for out-of-process MTS packages) whenever making interprocess calls. This made it impossible to enforce security policy properly in real-world applications deployed under IIS 4 unless the server box was dedicated to the purpose of hosting a single application and careful security configuration was performed by experienced system and network administrators. Even in optimal security configurations, the solutions to this and other IIS 4 and Windows NT Server security architecture limitations involved opening up intentional security holes on the internal network and crossing your fingers that no third party gained access to modify the content of your IIS 4-hosted application where they could take advantage of these intentional security holes. ASP.NET includes a `<processModel>` attribute for setting a custom COM impersonation level if you wish to change the default for IIS 5:

```
comImpersonationLevel =  
"[Default|Anonymous|Identify|Impersonate|Delegate]"
```

IIS 4 attempted to work around the limitations of COM impersonation through the dynamic caching of a thread's token when MTS-hosted components were created and behind-the-scenes conveyance to MTS of that cached token in order to reimpersonate the thread-impersonated identity during the interprocess COM call that would end up improperly impersonating IWAM_MachineName but the work-around did not stand up well to malicious code. It also created a dependency on the Single Threaded Apartment model for deploying server-side COM objects, which clearly was less than optimal for a robust heavily-utilized real-world environment.

System.Security

The common language runtime security system is implemented by classes that belong to the System.Security namespace. Everything that happens in ASP.NET is governed by these classes and the architecture for secure code execution that they provide. A core subset of System.Security, and one that has a direct bearing on all security features of ASP.NET, is an assortment of classes that enable .NET to set programmatically and identify automatically the effective user security context and evaluate the permissions that should be granted within that context. The core classes that provide this functionality and expose it within managed code implement one of three security class interfaces:

System.Security.Principal.IIdentity implemented by FormsIdentity, GenericIdentity, PassportIdentity, and WindowsIdentity

System.Security.Principal.IPrincipal implemented by GenericPrincipal and WindowsPrincipal

System.Security.IPermission implemented by CodeAccessPermission and PrincipalPermission

Each IIdentity derived class supports the three properties defined by the IIdentity interface, they are: AuthenticationType, IsAuthenticated, and Name. AuthenticationType is a String value representing the authentication type. IsAuthenticated is a Boolean value indicating whether an authentication operation has been performed successfully to authenticate the credentials provided by the user. Name is a String value reflecting the name of the user indicated by the IIdentity object. The Name property of any IIdentity object that has not been authenticated typically contains an empty value (""). When authenticated, Name includes a domain or host name prefix under AuthenticationType "NTLM" for Windows type authentication that relies on the NT Lan Manager network operating system.

IPrincipal includes as its only property, named Identity, an object that implements IIdentity. Its only method in addition to the IIdentity object it contains a reference to in its Identity property is a method named IsInRole which accepts a String parameter indicating the name of the role for which to check the Identity's membership status. If the IIdentity object referenced by the Identity property does belong to the specified role, the IsInRole method returns true. Otherwise IsInRole returns false.

IPermission is the foundation of permissions objects that exist within so-called evidence chains that are analyzed by code access permissions through a stack walk performed by instances of System.Security.CodeAccessPermission which implement both the IPermission and IStackWalk interfaces. A stack walk is performed to ensure that every caller that is responsible directly or indirectly for the invocation of a certain protected operation in fact has security permissions adequate to allow each caller to perform the operation. Instances of CodeAccessPermission that implement IPermission are central to the enforcement of code access security policy.

System.Security.Principal.IIdentity

IIdentity objects represent identities that are meaningful within a particular authentication scheme and authentication store. They are normally associated with IPrincipal objects that are able to determine role membership for the identity according to whatever role determination algorithm is appropriate for the identity. Classes that implement IIdentity also typically provide additional properties and methods that give information and abilities

to applications that enable user identification and decisions based upon the identities these objects represent.

Within any `IPrincipal` object in the object's `Identity` property you will find an object that implements `IIdentity`. Refer to `IPrincipal.Identity.Name` to get the name of the user identified by the `IIdentity` object. The structure of the `Name` value will vary depending on the value of the `IIdentity.AuthenticationType` property indicating the name of the authentication method used to validate credentials that prove code or user request to be authorized to act on behalf of the user account `Name`. After credentials have been validated the `IIdentity.IsAuthenticated` property is set equal to `true` by the authentication provider.

`System.Security.Principal.GenericIdentity`

`GenericIdentity` is a class that is useful for building your own authentication provider. `GenericIdentity` implements the `IIdentity` interface at its base level, with constructors that initialize the `Name` and `AuthenticationType` properties. Most applications will never use `GenericIdentity` directly, as creating a custom authentication provider is more work than simply customizing an existing `AuthenticationModule`. The other standard `IIdentity` classes: `FormsIdentity`, `PassportIdentity`, and `WindowsIdentity`, each derive from `IIdentity` directly not from `GenericIdentity`.

`System.Security.Principal.IPrincipal`

`System.Web.UI.Page`, the class from which every ASP.NET page is automatically inherited when compiled at run time on the server, contains a `User` property that obtains a reference to an object that implements the `IPrincipal` interface from the current `HttpContext` object under which the ASP.NET page request occurs. The `User` property of `HttpContext` supplies the user security context `IPrincipal` object that was attached to the request context by the `Authenticate` event as defined by the active `System.Web.Security` authentication module's event handler delegate.

The `IPrincipal` interface `IsInRole` method searches the list of roles that the attached `IIdentity` object is known to belong to and returns a `Boolean` value indicating whether or not the role name supplied to `IsInRole` is present in that list. The notion of roles varies in implementation depending upon the type of authentication used. Windows authentication relies on Windows group membership and roles map directly to individual groups. Forms authentication, Passport authentication, and client certificate authentication without certificate-to-Windows user account mappings established inside IIS provide no role functionality by default.

ASP.NET supports role-based security by providing the `IsInRole` method of the `IPrincipal` interface. This single method makes determining a user's effective role participation dramatically easier than in the past when Win32 API calls were necessary in order to extract a Windows user's membership in local or domain groups. Role-based security is an important aspect of ASP.NET. Security restrictions for users that belong to particular roles can be implemented either declaratively, at compile time, or imperatively, at run time. Declarative role-based security restrictions must be associated with a class,

method, attribute, or event declaration whereas imperative restrictions can be implemented as part of any run time code block.

To restrict access to a particular role in any ASP.NET application use the following declarative syntax in conjunction with class, method, attribute, and event declarations:

```
[PrincipalPermissionAttribute(SecurityAction.Demand, Role = "Authors")]
```

For example, a class named test can be restricted using role-based security to only allow users who belong to the "Authors" role to instantiate and use objects of the test class type with the following class declaration syntax:

```
[PrincipalPermissionAttribute(SecurityAction.Demand, Role = "Authors")]
```

```
public class test { public test() {}
```

```
public bool dotest() {return(true);}}
```

Any `IPrincipal` that returns false in response to `IsInRole("Authors")` will cause a `SecurityException` to be thrown at run time when an ASP.NET page attempts to instantiate an object of class type test. The same role-based security restriction can be created imperatively with the `System.Security.Permissions.PrincipalPermission` class:

```
public class test { public test() { System.Security.Permissions.PrincipalPermission p = new System.Security.Permissions.PrincipalPermission(null, "Authors"); p.Demand();} public bool dotest() {return(true);}}
```

The code shown here uses an instance of `PrincipalPermission` within the constructor of the test class. The `Demand` method is called to enforce the role-based security established as a required `Permission`; only members of the "Authors" role are permitted to execute the test constructor, all others will produce a `SecurityException`. The first parameter to the `PrincipalPermission` constructor can take a user name, while the second parameter is a role name. Declarative syntax also supports user name restriction with an added `Name` parameter that can appear in addition to or in place of the `Role` parameter to the declarative role-based security restriction.

System.Security.Principal.GenericPrincipal

`GenericPrincipal`, like `GenericIdentity`, is a class that is useful for building your own authentication provider. `GenericPrincipal` is also useful for extending the functionality of existing authentication providers. `GenericPrincipal` implements the `IPrincipal` interface at its base level, with a constructor that sets the `Identity` property and accepts an array of names of the roles to which that the given identity belongs. The `IsInRole` method of `GenericPrincipal` searches the list of role names provided in the constructor and returns true if the specified role is found in the list. The following code takes an existing `IIdentity` object created by some authentication provider and stores it inside a `GenericPrincipal` object that will return true for any `IsInRole` method call that passes the role name "Authors" as a parameter. The `Application_AuthenticateRequest` function is a default event handler inside the `global.asax` file.

```
protected void Application_AuthenticateRequest(Object sender, EventArgs e) {  
if(Context.User != null){Context.User = new GenericPrincipal(Context.User.Identity,new  
String[] {"Authors"}); }}
```

Context refers to the `HttpContext` property accessible from within `global.asax`. During processing of unauthenticated requests and when redirecting to a Forms authentication login page `Context.User` is null and therefore an if statement is required to avoid a runtime error prior to acquisition of an authenticated identity. Since the code shown only replaces the `IPrincipal` object that contains an `Identity` object already, and `IPrincipal`-derived classes don't typically implement other properties and methods beyond the `IsInRole` method and `Identity` property, replacing the `IPrincipal` object that existed before does not cause problems for an ASP.NET application. The other standard `IPrincipal` classes: `FormsPrincipal`, `PassportPrincipal`, and `WindowsPrincipal`, each derive from `IPrincipal` directly rather than using `GenericPrincipal` as a base class, but because of the shared `IPrincipal` interface implementation, instances of these classes can normally be used interchangeably in most applications.

Custom Dynamic Role Membership with a Custom `IPrincipal` Class

Using the instructions shown in this section you can implement custom `IPrincipal` classes that provide an application-specific role mapping algorithm. You've already seen how `GenericPrincipal` can be used to implement simple customized role name identifier lookups and set arbitrary role membership lists from a custom `Application_AuthenticateRequest` event handler in `global.asax`. This approach works well enough, but an application that needs to conduct a more complete dynamic review of a user identity's role participation, including the ability to detect changes to the list of roles to which the identity belongs or implement custom security policy such as "Employees" aren't allowed to use the ASP.NET application after 5:00pm or "Managers" who are not also "Certified Engineers" are not allowed to access engineering specifications. Creating a custom `IPrincipal` class is as simple as inheriting from `IPrincipal` and defining an accessor for the `Identity` property, creating a constructor, and providing an implementation of `IsInRole`.

```
public class MyPrincipal : IPrincipal {
    private Identity identity;
    public Identity Identity {get{return identity;}}
    public MyPrincipal(Identity i) {identity = i;}
    public bool IsInRole(string role){return(true);}}
```

The sample `IsInRole` method implementation shown here just returns true in response to every role requested. To put this `MyPrincipal` class to use, simply replace `GenericPrincipal` with `MyPrincipal` in an `Application_AuthenticateRequest` similar to the code shown in the previous section.

System.Web.Security

Security classes specific to ASP.NET are located in `System.Web.Security` in the .NET Framework class library. Table 5-1 lists the classes contained within the `System.Web.Security` namespace. Classes for ASP.NET authentication, authorization, and impersonation exist within this namespace that are complementary to the code access security, session management, and cryptography classes of ASP.NET's security architecture located in other namespaces. All of the classes in the `System.Web.Security` namespace are declared as `NotInheritable` (sealed in C#) so they cannot serve as base classes for custom derived classes.

Table 5-1: System.Web.Security Namespace

Class Name	Description
DefaultAuthenticationEventArgs	Default authentication parameters
DefaultAuthenticationModule	Default authentication module
FileAuthorizationModule	Automatic NTFS permissions module verifies ACL for impersonated user
FormsAuthentication	Authentication ticket utility class
FormsAuthenticationEventArgs	Forms authentication parameters
FormsAuthenticationModule	Forms authentication module
FormsAuthenticationTicket	Object wrapper around authentication ticket (cookie) in forms authentication
FormsIdentity	Identity object used by the FormsAuthenticationModule
PassportAuthenticationEventArgs	Passport authentication parameters
PassportAuthenticationModule	Passport authentication module
PassportIdentity	Identity object used by the PassportAuthenticationModule
UrlAuthorizationModule	URL permissions module that reads <authorization> section of .config
WindowsAuthenticationEventArgs	Windows authentication parameters
WindowsAuthenticationModule	Windows authentication module

The core purpose of the System.Web.Security namespace is to implement a system of delegates for layering-in event-driven custom application logic for carrying out authentication. Delegates, provide a mechanism for type-safe event programming and event parameter argument passing. The following delegates are part of the System.Web.Security namespace:

- DefaultAuthenticationEventHandler
- FormsAuthenticationEventHandler
- PassportAuthenticationEventHandler
- WindowsAuthenticationEventHandler

There is an event handler delegate for each authentication module in System.Web.Security. Of the members of System.Web.Security namespace, only the delegates and FormsAuthentication, FormsAuthenticationTicket, FormsIdentity, and PassportIdentity classes are meant to be used by typical ASP.NET application programmers. The rest of the classes exist to facilitate delegate functionality for wiring up events with delegated event handlers, passing and parsing type-safe event arguments.

Most developers will allow ASP.NET to instantiate and manage authentication and authorization modules, event handlers, and System.Security.Principal classes rather than customize these classes. Adding application-specific code to authentication event processing and setting or retrieving attributes of Forms- or Passport-authenticated user identities are the extent to which you will likely need System.Web.Security classes in your own applications.

Web Application Security

Security for Web applications in the past was a simple matter of configuring a user account database, filling it with user credentials, hooking up an authentication service provider layer implemented as an ISAPI filter DLL, disabling anonymous access to the files you wanted to password-protect, and crossing your fingers that buffer overflow vulnerabilities got discovered and patched by the good guys before the bad guys could find and exploit them. Those steps may seem overwhelming, especially if you don't know how to write ISAPI filters and access databases from C++ code, and that's why numerous third-party solutions are available plus Microsoft products Site Server or Commerce Server that provided ISAPI filters ready-made to plug-and-play.

Your application code could rely on the REMOTE_USER, AUTH_USER, and LOGON_USER ServerVariables to determine the user identity of the user making the request. Application-specific notions of groups and authorization permissions were up to the developer to add explicitly to the code. Impersonation was limited to IUSR_MachineName for anonymous impersonation or some other Windows user account set up as the authenticated user impersonation account. Once your code on the server had a LOGON_USER value to rely on, the rest was easy. The administrator would lock down IIS and OS security with the idea that only the two impersonation accounts would ever execute code on behalf of Web site users, and there was nothing at all to worry about. At least nothing over which you have control, and worrying about things over which you have no control is often a waste of time.

With ASP.NET you have much less to worry about. You don't have to worry about buying another product to get an ISAPI filter DLL that connects your user credentials database with the authentication layer to enforce password protection. You don't have to worry about bugs in cookie-based authentication schemes because the pain caused by those bugs in the past (especially in Microsoft Site Server) has led to properly-designed cryptographically-secured session-based authentication schemes implemented by ASP.NET Forms and .NET Passport authentication. And you don't have to worry about coding your own ISAPI DLLs and COM objects to implement high-performance services to include database connectivity and network-awareness.

Understanding Insecure Authenticated Sessions

ASP.NET lets you worry only about your application's requirements and it lets you express those requirements in terms of low-level security policies. Hopefully the opportunity to focus your development efforts on creating application logic rather than struggling to understand security flaws in systems that aren't supposed to be vulnerable to them to begin with will decrease the number of times that you write code that does the following brain-dead operation:

1. Check for cookie
2. If no cookie exists, generate unique number and drop it as a cookie
3. Accept user input and store it in session state keyed by unique number cookie
4. Display to the user in a dynamically-constructed Web page everything contained in session state for the session identified by the unique number
5. Repeat steps 1 through 4 until the user finishes whatever they're doing

This 5-step sequence is, unfortunately, quite common in Web applications built using older development environments. Never do this. It exposes the contents of your users' sessions to anyone who can write a program that sends an HTTP request with a variable cookie header. The worst thing you can do with a Web application is create code that performs these 5 steps and then lets the user authenticate with the server by providing credentials that are validated against an authentication store whereupon your application sets a flag in session state indicating that the user has successfully authenticated and the session should thereafter be allowed access to all information the user has provided during past sessions.

Web applications that do this also tend to give these "authenticated sessions" permission to access restricted areas of the site, change passwords, and perform actions reserved for trusted users. You may think it's obvious that a unique number assigned arbitrarily to a client as a cookie is inadequate authentication, but legions of Web developers don't see it that way, since only users who can provide valid credentials end up with authenticated sessions many Web developers think that an authenticated session is more secure than an unauthenticated session. The truth is that only requests that supply a shared secret or supply a valid client certificate can be treated as authenticated requests. Not only can random number attacks penetrate authenticated sessions easily, more importantly the network itself assists users in hijacking sessions when they share proxy servers that cache responses to HTTP GET requests aggressively. Knowledge Base article Q263730 details the way in which proxy servers are known to cache Set-Cookie headers and inappropriately issue these headers to multiple users.

ASP.NET is able to use cookies for reliable and secure authenticated sessions because they are constructed and issued in a cryptographically-secure manner and automatically expire according to configurable timeout settings. Forms authentication automates this process to provide protection against malicious attacks that use random cookies and protection against known flaws in cookie-based session state tracking. To limit the damage caused by eavesdroppers, Forms authentication generates a shared secret called a Forms authentication ticket that it sets in a cookie on the client by issuing a Set-Cookie after an authentication event that can optionally be SSL-encrypted. Further, ASP.NET does not incorrectly mix session state and authentication as do other systems that provide cryptographically-invalid "authenticated sessions".

Because the Set-Cookie header is issued by ASP.NET only in response to an HTTP POST that includes valid authentication credentials, problems caused by aggressive proxy servers as documented in Knowledge Base article Q263730 are avoided. Only a POST request that includes the same credentials in the request body will potentially result in duplicative Set-Cookies served out of proxy cache. This could result in a user behind a proxy server being unable to logout and log back in to an ASP.NET application until proxy cache flushes the expired Forms authentication ticket but it will not result in a user receiving another user's authentication ticket.

Chapter 6: ASP.NET Application Security

Security settings for ASP.NET applications are included in the web.config and machine.config files. In addition to these two configuration files there are specialized security configuration files for the .NET Framework that follow the same XML structure using specialized XML schemas and define code access security as well as machine-specific or enterprise-wide security policies. The Config directory inside the root install location of the .NET Framework runtime is the home of Enterprisesec.config and Security.config in which .NET Framework security policy settings are stored.

ASP.NET's script engine, aspnet_isapi.dll, must be associated with the .config file extension inside Application Mappings for any Web application that contains .config files, otherwise IIS will not protect .config files from access by Web browsers. As long as aspnet_isapi.dll is configured as the script engine responsible for processing requests for .config files it will return HTTP error 403 access forbidden in response to incoming requests for .config files.

The security-related elements of ASP.NET's configuration file XML schema are contained within <system.web> and include <authentication>, <authorization>, and <identity> as shown here in template form. Each of the <authorization> elements <allow> and <deny> accept multiple users, roles, and verbs as comma-separated lists.

```
<system.web>
<authentication mode="[Windows | Forms | Passport | None]">
  <forms name="name"
  loginUrl="url"
  protection="[All | None | Encryption | Validation]"
  timeout="30" path="/">
  <credentials passwordFormat="[Clear | SHA1 | MD5]">
  <user name="username" password="password" />
  </credentials>
  </forms>
  <passport redirectUrl="internal"/>
</authentication>
<authorization>
  <allow users="[* | ? | names]" roles="[roles]"
  verbs="[verbs]"/>
  <deny users="[* | ? | names]" roles="[roles]"
  verbs="[verbs]"/>
</authorization>
<identity impersonate="[true | false]" userName=""
password=""/>
</system.web>
```

Internet Information Services' authentication always takes priority over any ASP.NET authentication because IIS won't load aspnet_isapi.dll and hand off request processing to it for an ASP.NET application file until after the request has authenticated successfully with IIS. No authentication is performed by IIS when Anonymous access is enabled as

shown in Figure 6-1 for IIS 5 and Figure 6-2 for IIS 6. Impersonation and authorization still occur, of course, and the user security context under which all Anonymous requests are processed by IIS is determined by the setting Account used for anonymous access as shown in Figures 6-1 and 6-2.

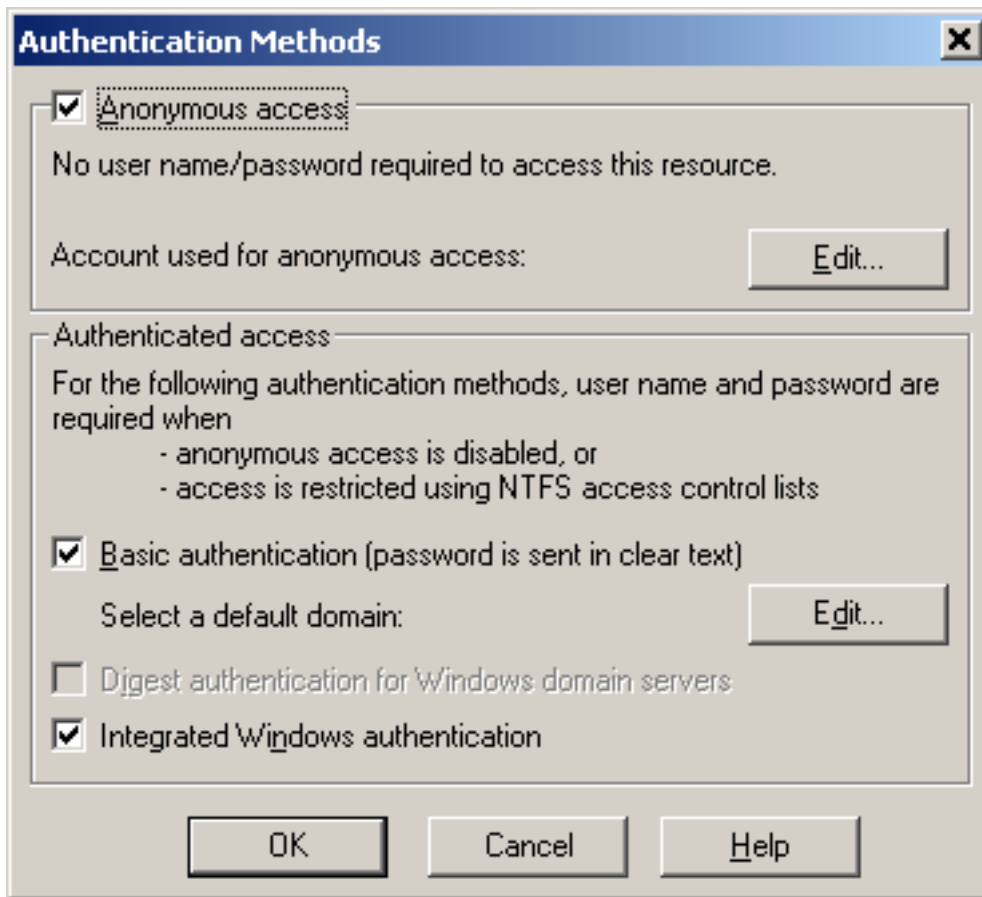


Figure 6-1: Anonymous access is enabled for IIS 5 in the Directory Security properties tab

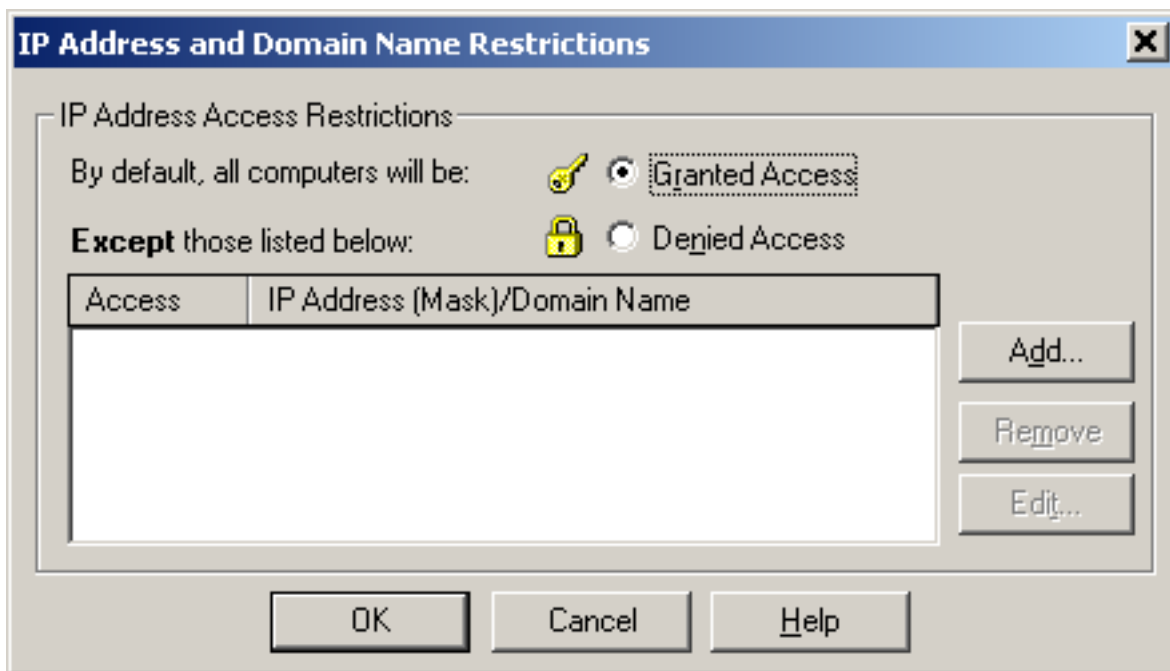


Figure 6-2: Anonymous access is enabled for IIS 6 in the Directory Security properties tab

IIS attempts to open the file requested by the Web client using the Anonymous user impersonation account security context, which by default is IUSR_MachineName where MachineName is the network name assigned to the server computer, and NTFS ACL permissions are checked for read access rights by the OS. In this way Anonymous impersonation and authorization are implemented by IIS before aspnet_isapi.dll is given an opportunity to take over request processing.

Anonymous access is enabled through the Authentication Methods window which is opened by clicking the Edit button under Anonymous access and authentication control inside the Directory Security properties tab. Figure 6-3 shows the location of the Edit button within Directory Security.

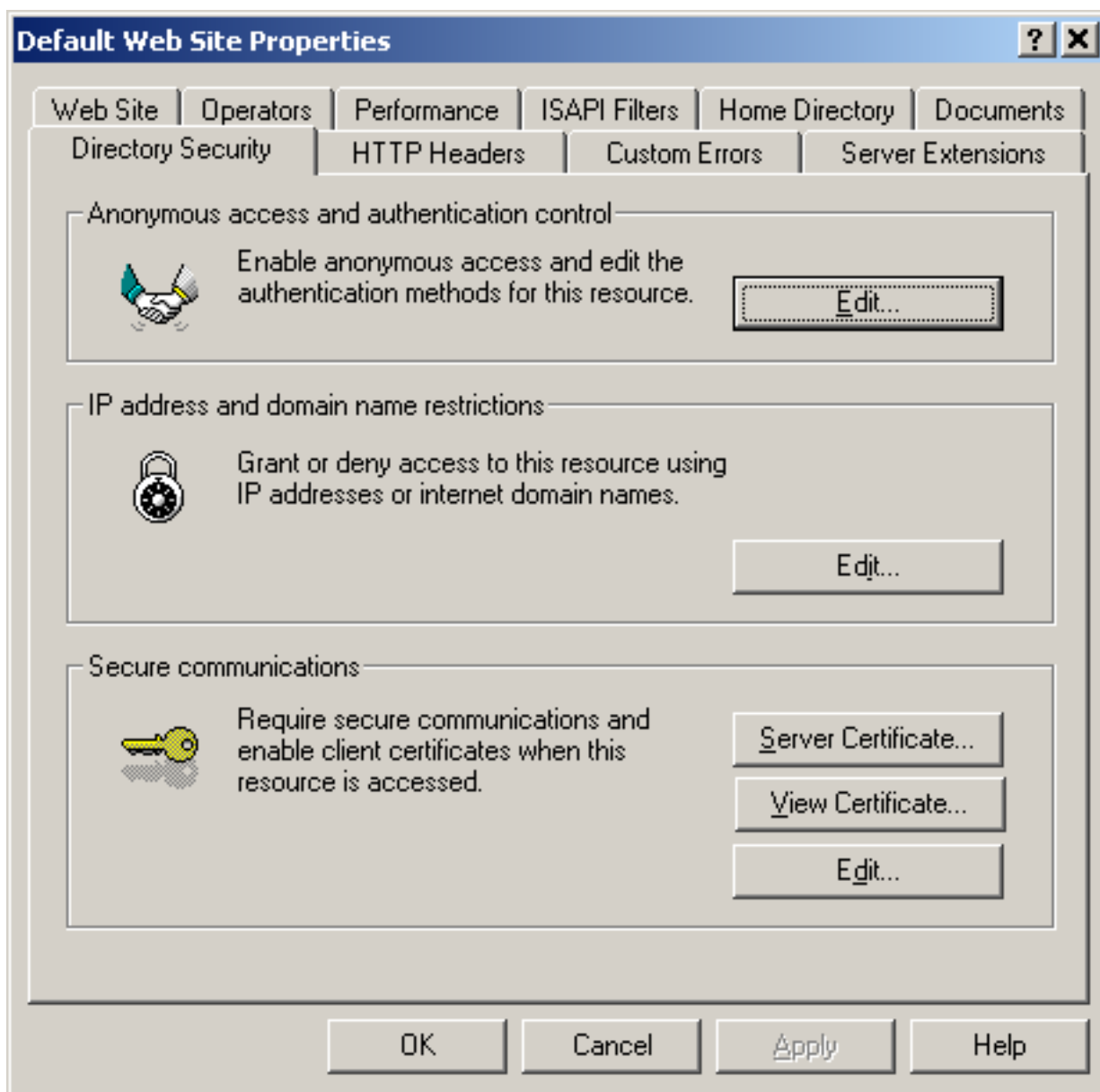


Figure 6-3: Directory Security Tab Enables Editing of IIS Application Security Properties

Additionally, IIS are able to grant or deny access to a requestor based on the IP address from which the request originated. Internet domain name restrictions can also be configured by telling IIS to perform a DNS reverse-lookup on the requestor's IP address and use the result to evaluate access permissions. Figure 6-3 also shows the location of the IP address and domain name restrictions Edit button.

When you elect to restrict access by domain name, IIS displays the following alert: "Warning: Restricting access by domain name requires a DNS reverse lookup on each connection. This is a very expensive operation and will dramatically affect server performance." The performance impact of this security measure will only matter if you are hosting a busy Web site visited by members of the public. If your Web site is visited only by members of a particular domain or group of domains and you aren't servicing tens of thousands of hits per minute, the security benefit outweighs the slight performance decrease. In this mode a hacker would have to simultaneously hijack the DNS server that your server

contacts for name resolution or supply origination IP addresses that resolve to authorized domains in order to access your Web applications. However, this setting creates a direct dependency on a DNS server for request processing; if the DNS server goes down, request processing will stop. This option is not suitable for Web sites that receive visitors from the general public because many IP addresses cannot be resolved to a domain name through a DNS reverse lookup.

Using ASP.NET Authentication

There are two parts to network authentication in ASP.NET: the authentication provider and an authentication data store. The authentication provider is responsible for intercepting the requestor's credentials using a particular authentication protocol. An authentication data store is a repository of valid credentials or credential hash codes against which the requestor's credentials might be validated. ASP.NET provides the following authentication providers:

- Windows Integrated (defers to IIS)
- .NET Passport (uses passport.com)
- Forms (login via custom HTML FORM)

When Windows integrated authentication is used, the authentication setting of IIS controls authentication for access to files and directories served by IIS. Because every ASP.NET application is served by IIS, the security settings in IIS can be used to control access to ASP.NET applications. IIS versions 5 and 6 support the following authentication modes:

- HTTP Basic
- HTTP Digest
- Windows NT LAN Manager (NTLM/ Kerberos)
- Client Certificates
- Integrated .NET Passport (IIS version 6 only)

IIS authentication settings are enforced regardless of ASP.NET application specific configuration. When IIS requires authentication in addition to ASP.NET the user may be required to authenticate twice to gain access to an application.

Each authentication provider offers support for a different default authentication data store. Only Forms authentication can be connected to any authentication store through the addition of application-specific code. An authentication store that keeps only a hash code value computed from valid credentials or that is able to compute the validity of credentials dynamically, as with Client Certificates where the digital signature of a Certification Authority (CA) is validated when the client certificate is offered as a credential, is superior to an authentication store that keeps a complete copy of each credential. The reason is that any code that can read from the authentication store to obtain complete credentials can successfully penetrate any aspect of such a system's security. Whereas reading from an authentication store that does not contain complete credentials but rather contains only enough information to validate credentials when supplied for authentication is inadequate to compromise system security; such systems can only be penetrated by code that is able to write to the authentication store or intercept valid credentials when

they are transmitted over the network and received and validated by the authentication provider.

The setting inside machine.config or web.config that configures authentication for ASP.NET is <authentication> which appears inside <system.web> as shown. The default machine.config file installed with ASP.NET sets authentication mode to Windows. If your ASP.NET application needs to use a different authentication setting you enter it into web.config. Each ASP.NET application can have only one authentication mode for all of its subdirectories, so only web.config in the application root directory has any effect.

Application subdirectories can't use an authentication mode different from the application root and other subdirectories.

```
<!--  
authentication Attributes:  
mode="[Windows|Forms|Passport|None]"  
-->  
<authentication mode="Windows">  
</authentication>
```

Any request that provides credentials that are validated against an authentication store can be considered fully authenticated and be given any level of trust that is appropriate for the authenticated identity. Windows authentication, with its built-in authentication store (Windows local or domain accounts), built-in authorization controls (NTFS, local and domain security policy) and use of HTTP Basic or Digest authentication or Windows NT Lan Manager (NTLM) provides the simplest mechanism for achieving full trust in ASP.NET authentication.

Forms and Passport authentication both rely on cookies for chaining together requests subsequent to an authentication event. The client authenticates by providing valid credentials to the server and the server passes back a shared secret that the client uses to authenticate subsequent requests. Forms and Passport cookies are an adequate mechanism for establishing complete trust without the security risk inherent to passing full credentials over the network unencrypted in each request. For ideal security SSL must be used to encrypt all communication between the client and server. Many sites opt to use Forms and Passport authentication and to SSL-secure only the login step where authentication credentials are passed to the server. Subsequent requests are authenticated by the shared secret passed by the client through cookies, and an eavesdropper who intercepts the cookie does not end up in possession of the full authentication credentials that would allow them to authenticate successfully at any time and potentially with other servers as well.

System.Web.Security.WindowsAuthenticationModule

With Windows authentication, HTTP authentication headers include the user ID and password credentials with each request to the password-protected items and no code needs to be written in order to implement and enforce security policy. By combining Windows authentication with ASP.NET Impersonation, NTFS permissions and Windows user accounts enforce security policy automatically.

All Windows authentication provided by ASP.NET when <authentication mode="Windows"> is implemented by the WindowsAuthenticationModule class. This is not a class you will ever use directly in your application code, it exists only to be instantiated by ASP.NET as one of the configured <httpModules>. The class System.Web.Security.WindowsAuthenticationModule implements the System.Web.IHttpModule interface and like other such modules it is loaded into the Modules property of the System.Web.HttpApplication derived class instance that represents the ASP.NET application base class at run time. The module instance can be accessed using the following type-cast reference from within any ASP.NET page:

```
(WindowsAuthenticationModule)Context.ApplicationInstance.Modules["WindowsAuthenticatio  
n"]
```

Context references the System.Web.UI.Page class HttpContext property. Within Context is a reference to the HttpApplication-derived class created automatically by ASP.NET for the current application or derived explicitly in global.asax from the System.Web.UI.Page class. ASP.NET places an instance of each IHttpModule class listed in <httpModules> in the Modules collection of the HttpApplication-derived object that exists for processing of the page request.

WindowsAuthenticationModule hooks up event handlers using the delegate declared as System.Web.Security.WindowsAuthenticationEventHandler for handling its Authenticate event which wires up an event handler chain that includes by default the System.Web.HttpApplication.AuthenticateRequest event that your application code can handle by overriding a default event within an HttpApplication derived base class for the ASP.NET application as defined inside the global.asax file. The WindowsAuthenticationModule automatically adds a delegate to the AuthenticateRequest event named WindowsAuthentication_Authenticate. Define this default event handler function as part of your HttpApplication derived base class within global.asax and your application code will be called as part of the event chain.

```
protected void WindowsAuthentication_Authenticate(  
Object sender, WindowsAuthenticationEventArgs e) {}
```

By type casting the WindowsAuthenticationModule object reference found inside the Modules collection you can access in your code the Authenticate event for the purpose of adding additional EventHandler delegates or removing the event handler delegate added automatically by WindowsAuthenticationModule. For example, the following global.asax will result in removal of the event handler delegate established by the WindowsAuthenticationModule and replace it with a custom delegate called myHandler. During processing of the Authenticate event under <authentication mode="Windows"> only the new myHandler delegate will be called.

```
<%@ Import namespace="System.Security.Principal" %>  
<%@ Import namespace="System.Web.Security" %>  
<Script language="C#" runat="server">  
myHandlerClass mhc = null;  
protected void Application_BeginRequest(Object sender, EventArgs e) {
```

```

if(mhc == null) {
mhc = new myHandlerClass();
((WindowsAuthenticationModule)Modules["WindowsAuthentication"]).Authenticate -= new
    WindowsAuthenticationEventHandler(WindowsAuthentication_Authenticate);
((WindowsAuthenticationModule)Modules["WindowsAuthentication"]).Authenticate += new
    WindowsAuthenticationEventHandler(mhc.myHandler); }}}
public class myHandlerClass {
public void myHandler(System.Object sender, WindowsAuthenticationEventArgs e) {
e.Context.Response.Write("<p>myHandler</p>"); }
</script>

```

The myHandler function uses the Context property of WindowsAuthenticationEventArgs to access the Response object and write a single HTML paragraph. Other properties of WindowsAuthenticationEventArgs include Identity and User which are used to read the WindowsIdentity object and read or set the IPincipal object associated with the current request.

System.Security.Principal.WindowsIdentity

WindowsAuthenticationModule implements default functionality for Windows user account authentication, authorization, and impersonation. It creates an instance of System.Security.Principal.WindowsIdentity and attaches it to an instance of System.Security.Principal.WindowsPrincipal within HttpContext.User where the User security context is stored for the current request. WindowsIdentity objects are valuable at runtime for their IsAnonymous, IsGuest, IsSystem, and Token extended properties that aren't available from other built-in IIdentity classes.

WindowsIdentity also includes a method named Impersonate that enables your code to switch temporarily to a different Windows user security context. It also includes a static method named GetCurrent that returns a WindowsIdentity object that represents the current Windows user security context. Another static method named GetAnonymous returns a WindowsIdentity object initialized to the null state of an anonymous user security context that contains empty user name and security token.

To force a thread's security context to reflect a different Windows user account you need to obtain a platform-specific handle to the user's security token. The ADVAPI32.DLL contains a LogonUser function that enables this through a call to native code if you configure .NET security policy to allow native code access. Using the token handle you can create a new WindowsIdentity object that represents the Windows identity then call ImpersonateIdentity to switch the calling thread's effective security context to that specified by the WindowsIdentity. The Impersonate method returns a WindowsImpersonationContext object with an Undo method that is used to revert the thread to the previous impersonation identity on demand. By default, any ASP.NET code can impersonate the security context of the host process using the following code that takes advantage of a flaw in ASP.NET Impersonation to force the thread to use the host process token security context.

```

WindowsIdentity id = WindowsIdentity.GetCurrent();
Response.Write("<p>Current User: " + id.Name + "</p>");

```

```

WindowsIdentity id2 = new WindowsIdentity(new IntPtr(1));
WindowsImpersonationContext idcontext = id2.Impersonate();
WindowsIdentity id3 = WindowsIdentity.GetCurrent();
Response.Write("<p>Current User: " + id3.Name + "</p>");
idcontext.Undo();
WindowsIdentity id4 = WindowsIdentity.GetCurrent();
Response.Write("<p>Current User: " + id4.Name + "</p>");

```

The code shown calls Undo on the WindowsImpersonationContext to revert back to the original user security context. This code produces output like the following, where IUSR_MachineName is the Anonymous impersonation account and DOMAIN is a name of the server with ASP.NET installed or its Windows domain:

```

Current User: DOMAIN\IUSR_MachineName
Current User: DOMAIN\ASPNET
Current User: DOMAIN\IUSR_MachineName

```

In an ideal world, ASP.NET would not have been designed to allow any thread with any security context to arbitrarily impersonate the security context of its host process. The default configuration of ASP.NET is unfortunately not secure in this respect. The default security policy setting in a future ASP.NET service pack may plug this security hole. To prevent this behavior, you can override the default Code Access Security policy trust level, Full, and set it instead at a lower trust level. The machine.config file contains a <trust> directive to control this setting as shown.

```

<securityPolicy>
<trustLevel name="Full" policyFile="internal"/>
<trustLevel name="High" policyFile="web_hightrust.config"/>
<trustLevel name="Low" policyFile="web_lowtrust.config"/>
<trustLevel name="None" policyFile="web_notrust.config"/>
</securityPolicy>
<!-- level="[Full|High|Low|None]" -->
<trust level="Full" originUrl="" />

```

Change <trust level> to High, Low, or None in order to further restrict the default Code Access Security policy trust level for ASP.NET applications hosted under IIS on your server box. For details on the trustLevel policy configuration read the .config file for each of the trustLevels defined by machine.config as shown.

System.Security.Principal.WindowsPrincipal

The WindowsPrincipal class implements the logic necessary for the IPrincipal derived object to associate the encapsulated IIdentity object with a Windows user account and perform the Win32 API calls necessary to look up local or domain group membership. The IsInRole method of WindowsPrincipal is overloaded to include variations that accept numeric Role Identifiers and members of the enumeration System.Security.Principal.WindowsBuiltInRole in addition to String role names. WindowsBuiltInRole enumeration members include AccountOperator, Administrator, BackupOperator, Guest, PowerUser, PrintOperator, Replicator, SystemOperator, and User. The ability to work with role designators from WindowsBuiltInRole makes the

WindowsPrincipal class especially useful for ASP.NET applications that use Windows user accounts for authentication and authorization purposes.

.NET Passport

Microsoft .NET Passport is a single sign-in service that enables third-party Web sites to authenticate users through the .NET Passport authentication store. Users provide Passport credentials to Microsoft and the third-party Web site is informed by the Passport service that the credentials provided by the user were authentic so that the Web site can proceed with authorization and the implementation of appropriate security policy for the authenticated Passport user identity.

Each ASP.NET application's web.config file can include a <passport> directive listing the URL to which the client browser will be redirected automatically when a user who has not authenticated with .NET Passport attempts to access a restricted URL in the ASP.NET application. The following is the default <passport> value.

```
<!--  
passport Attributes:  
redirectUrl=["url"] - Specifies the page to redirect to, if the page requires authentication, and  
the user has not signed on with passport  
-->  
<passport redirectUrl="internal"/>
```

Microsoft's Passport service should never be used by anyone other than Microsoft. Its design is extremely inappropriate to the goal of privacy, and it centralizes user identity profiles and creates a single point of failure for authentication credential theft where the real impact of successful penetration is amplified in magnitude the more sites there are that use Passport. There is no reason to use Passport in your own site, so do not do so.

Forms Authentication

Forms authentication uses an HTML form to allow the user to enter a user ID and password as authentication credentials. Forms authentication then relies on a cookie-encoded Forms authentication identifier called a ticket that includes the authenticated user's login ID to enable ASP.NET to determine the authenticated identity in subsequent requests. Optionally, to provide better privacy protection, the contents of the Forms authentication ticket can be encoded using a cryptographic hash. Because HTTP is stateless, each request must be authenticated anew. An authentication ticket provides protection against hackers who might otherwise be able to guess valid session identifiers.

The .NET Framework documentation uses some unclear terminology to explain certain aspects of Forms authentication and you must rationalize what you read in the documentation in order to apply Forms authentication securely. Among other technical terminology prevalent in the documentation you will find cookies incorrectly referred to as "forms". Mistakes like these in the documentation make it difficult to understand ASP.NET Forms authentication and determine for yourself that the mechanism of ticket cookies used by Forms authentication is secure and reliable enough to use in production systems.

Forms authentication is considered safe and secure because a ticket is generated by the server and included in the Set-Cookie HTTP header sent to the client in response to an HTTP POST. The POST request includes valid credentials encoded in the body and known problems with proxy cache and cookies therefore do not apply to the cookie dropped by ASP.NET Forms authentication. No proxy servers cache the result of POST operations without regard for the contents of the request body, so even if the Set-Cookie is cached by a proxy it will only be given out to a client browser that POSTs the same valid credentials in the request body as did a previous request.

Microsoft Knowledge Base article Q263730 details cookies authentication flaw

There is a possibility that this scenario will result in a user's inability to log back in to your ASP.NET application after logging out since ASP.NET will invalidate the old ticket and attempt to issue a new one. The client may receive the old ticket from proxy cache rather than the new one and the result will be the appearance of an authentication failure. But the false positive authentication that is known to occur with cookies that are used as authentication in systems like Microsoft Site Server will not happen with ASP.NET Forms authentication. Note, however, that your own code can be susceptible to this flaw, as can any type of anonymous user session state that is tracked by cookies or session identifiers delivered to clients in response to HTTP GET requests.

ASP.NET uses the Forms authentication ticket to associate requests with a previous authentication event where credentials were provided by the client and validated against an authentication store. This ticket is enough to consider the subsequent requests to be authenticated because the ticket is only provided to a client after credentials are validated.

The ticket is more than a better session identifier because it is lengthier than most session identifiers and therefore more difficult to guess and it is constructed using a cryptographic Message Authorization Code. But you do not know for sure that the subsequent request in which the cookie is provided truly comes from the user that provided the valid authentication credentials previously. More to the point, you don't have any reason to believe that the user making the subsequent request in fact knows any valid authentication credentials because credentials are not supplied with the subsequent request.

Even when SSL encryption is used to secure all communication with the client you must assume that the authentication ticket has been compromised by an eavesdropper or a hacker and reauthenticate full credentials prior to taking any action in your application that reveals or modifies sensitive data. For example, it would be inappropriate for your application to allow the user to change a password without first asking the user to provide the original password again for confirmation. Skipping this reauthentication step is not acceptable as a time-saving shortcut because it renders your server vulnerable to penetration by any eavesdropper who intercepts an authentication ticket. It also leaves your server vulnerable to brute force attacks because authentication tickets can be guessed randomly by hackers with the help of cracking software. Any authentication ticket that is valid for your application should be afforded limited rights and each operation that reveals or modifies sensitive data should refuse to accept a ticket in place of credentials. Require definitive authentication through reverification of the user's

credentials instead. If your application enables many sensitive operations to be performed by every user, require SSL at all times and require periodic reauthentication, especially prior to performing critically sensitive operations like a password change.

Forms authentication is configured for an ASP.NET application through the following <forms> directive inside an application root web.config file. The <forms> XML element appears within <authentication> where mode="Windows" as shown.

```
<authentication mode="[Windows | Forms | Passport | None]">
<!--
forms Attributes:
name="[cookie name]" - Name of the cookie used for Forms Authentication
loginUrl="[url]" - Url to redirect client to for Authentication
protection="[All|None|Encryption|Validation]" - Protection mode for data in cookie
timeout="[seconds]" - Duration of time for cookie to be valid (reset on each request)
path="/" - Sets the path for _the cookie
-->
<forms name=".ASPXAUTH" loginUrl="login.aspx" protection="All" timeout="30" path="/">
<!--
credentials Attributes:
passwordFormat="[Clear|SHA1|MD5]" - format of user password value stored in <user>
-->
<credentials passwordFormat="SHA1">
<user name="UserName" password="password"/>
</credentials>
</forms>
</authentication>
```

The name attribute of <forms> element specifies the name of the cookie set on the client by the Set-Cookie HTTP header that the Forms authentication module uses to send the Forms authentication ticket to the client. The loginUrl attribute specifies the URL to which unauthenticated requests will be redirected by ASP.NET when a client browser requests a URL that is password-protected in the application. User accounts can be added to the <credentials> element as <user> tags. Chapter 12 shows how to dynamically add <user> accounts to <credentials> using ASP.NET code.

Authorization of Request Permissions

The burden of deciding authorization permissions for each user lies with your application code when you use Forms authentication or Passport authentication without Windows account mapping because there is no user-specific Windows security context available for ASP.NET to use to evaluate NTFS permissions. In order to rely on Windows user accounts and NTFS permissions for authorization Windows authentication must be used or Passport authentication with Windows account mapping must be active.

ASP.NET performs Authorization of each request both explicitly and implicitly to varying degrees depending upon the configuration settings for authorization in each application. After an authenticated identity has been established, even if that identity is just the anonymous user identity, ASP.NET performs two automatic authorizations:

File Authorization URL Authorization

File Authorization uses `System.Web.Security.FileAuthorizationModule` and URL authorization uses `System.Web.Security.UrlAuthorizationModule`. Both authorizations rely on the ability to match an authenticated identity with matching entries in a permissions list. Many applications will need to implement custom authorization permissions

File Authorization

`FileAuthorizationModule` implements filesystem (normally NTFS) permissions evaluation for the authenticated user identity. This is most useful in conjunction with Windows authentication, as the effective user identity for the authenticated Windows user account will be meaningful within the context of an NTFS Access Control List (ACL). All other authentication types rely on the anonymous identity, which normally translates to `IUSR_MachineName` unless a different anonymous user account is configured in IIS, and use authenticated user identifiers that do not correspond to native Windows user accounts and which do not represent distinct identities at the level of code execution. For anonymous requests the `FileAuthorizationModule` simply confirms that `IUSR_MachineName` has appropriate permission to perform the requested operation on the specified files. For authenticated requests that do not use Windows authentication the authorization performed by `FileAuthorizationModule` can be supplemented with additional operating system-level security authorization through the use of the ASP.NET Impersonation feature.

URL Authorization

ASP.NET implements through `UrlAuthorizationModule` a system of user- and role-based URL permissions in addition to those implemented by IIS and application code. Using elements of the `<authorization>` section of a `web.config` file, optionally wrapped within a `<location>`, you can configure application-specific URL permissions for authenticated user identities. Two elements provide the ability to programmatically allow or deny access to any `<location>` within the URI namespace of the ASP.NET application: `<allow>` and `<deny>`. Since this authorization feature is provided by ASP.NET, access only to file types that are associated with `aspnet_isapi.dll` in the Application Mappings of IIS will be impacted by these `<authorization>` settings as discussed previously in this Chapter. The `<allow>` and `<deny>` elements take the following form:

```
<authorization>  
<allow users="list" roles="list" verbs="list" />  
<deny users="list" roles="list" verbs="list" />  
</authorization>
```

Either or both of “users” and “roles” must be specified, and each list enclosed in quotes is a comma-separated set of values; a user or group list for users and roles, and a list of HTTP methods (GET, HEAD, POST, DEBUG) to allow or deny for the optional verbs parameter. Two special values are supported for inclusion in the users list: “?”, which stands for the anonymous user identity, and “*”, which stands for all user identities. A

web.config file containing <allow> and <deny> elements can be placed in any directory of your ASP.NET application to establish for the directory and its subdirectories a common set of URL permissions. Alternatively, <location> can be used to specify <authorization> settings for a particular file or subdirectory. The following example allows users Jason and Harold and denies anonymous access:

```
<authorization>
<allow users="Jason, Harold" verbs="GET, POST" />
<deny users="*" verbs="GET, HEAD, POST, DEBUG" />
</authorization>
```

For Windows authentication, users within particular domains managed by a Windows NT or active directory domain controller can be listed explicitly by including the domain identifier as part of the listed user identity. The following example shows the use of a <location> element to allow access to a given subdirectory by a particular user in a particular Windows domain:

```
<location path="subdirectory">
<authorization>
<allow users="DOMAIN\Ralph" verbs="GET, POST" />
</authorization>
</location>
```

The default machine.config installed with ASP.NET includes <allow users="*" /> and each Web site or application must define more restrictive permissions with a <deny> element in order to override the machine.config if it is left in its default configuration. If any <deny> element applies to a given request, an HTTP 401 unauthorized error is returned to the requestor.

Custom Authorization with Application Code

Many applications need to impose additional authorization logic for each request. ASP.NET enables this through the Application_AuthorizeRequest event handler wired up in global.asax by default. The following code shows a simple event handler implementation that prevents users who are not located on the local host from being authorized by ASP.NET to receive responses to HTTP requests.

```
<Script language="C#" runat="server">
protected void Application_AuthorizeRequest(Object sender,
EventArgs e) {
if(Request.UserHostAddress != "127.0.0.1"){CompleteRequest();}}
</script>
```

This sample code uses the CompleteRequest method which is a member of HttpApplication, the base class for all ASP.NET applications. It causes ASP.NET to bypass the remaining events in the HTTP request processing event chain and go right to the last event, EndRequest. To throw an exception instead of ending request processing after performing application specific authorization you can use the Context.AddError method as follows.

```
Context.AddError(new Exception("Unauthorized"));
```

The Exception is added to the Exception collection for the current HTTP request and if it goes unhandled causes ASP.NET to display the exception to the user. Normally there is no opportunity for your code to handle the exception raised in this way, so adding an exception to this collection causes request processing to terminate abruptly in most circumstances. Only if you've layered-in your own exception handlers will Context.AddError result in a second chance for your code to touch the HTTP request processing and potentially handle the exception.

ASP.NET Impersonation and No-Code Authorization

Impersonation is the process by which an ASP.NET application receives a Windows user account token under whose security context to execute. Understanding impersonation in ASP.NET is more difficult if you have previous experience with ASP and Internet Information Services because impersonation terminology is applied differently now than it was in the past. With ASP.NET you can switch impersonation on and off, override the impersonation identity provided by IIS, switch temporarily to other identities during code execution, and map user identities contained within a non-Windows domain authentication store to Windows domain or local system user accounts. You can control access permissions on your Windows system and your internal network using active directory or local security policy without the security risk of transmitting a Windows user ID and password over the Internet.

Anonymous requests and requests processed with ASP.NET impersonation turned off use the ASPNET user identity. Depending upon your installation this user identity will either be a local account or a domain/active directory account. The following ASP.NET C# code determines the current Windows identity using WindowsIdentity.GetCurrent to display its name and platform-specific token handle:

```
<%@ Page language="C#" %>  
<%@ Import namespace="System.Security.Principal" %>  
<% WindowsIdentity id = WindowsIdentity.GetCurrent();  
Response.Write("<html><body><h1>" + id.Name);  
Response.Write("</h1><h2>" + id.Token);  
Response.Write("</h2></body></html>"); %>
```

Impersonation begins at the operating system level with a user security context setting for the executable code that created the current process. Typically the executable code is hosted by dllhost.exe (IIS 5) or w3wp.exe (IIS 6) and the process created through execution of these host modules is determined by IIS configuration settings for either the out of process application (IIS 5) or Application Pool (IIS 6). When ASP.NET script engine is invoked within dllhost.exe (or inetinfo.exe for in-process applications) it uses the following <processModel> configuration element found in machine.config to determine the user security context under which to run.

```
<!--  
processModel Attributes:
```

enable="[true|false]" - Enable processModel
userName="[user]" - Windows user_to run the process as.
Special users: "SYSTEM": run as localsystem (high privilege admin) account.
"machine": run as low privilege user account named "ASPNET".
Other users: If domain is not specified, current machine name is assumed to be the domain name.

-->

```
<processModel enable="true" userName="machine">
```

When hosted under IIS 6 within w3wp.exe instances, ASP.NET ignores the <processModel> settings found in the machine.config file, hosting itself instead within the "NT AUTHORITY\NETWORK SERVICE" built-in security context. If userName="machine" then ASP.NET is invoked by IIS 5 inside a dllhost.exe process whose security token is that of the "ASPNET" user account. Under IIS 5 you can also optionally force IIS to execute dllhost.exe using the SYSTEM security context by setting userName="SYSTEM".

To enable impersonation so that a user security context and token other than that of the ASP.NET user account "ASPNET", privileged "SYSTEM" account, or "NT AUTHORITY\NETWORK SERVICE" is used to process requests for users' authenticated identities, you set the <identity impersonate> setting to "true" inside machine.config or web.config. By default impersonate is set to "false" as shown here:

<!--

identity Attributes:

impersonate="[true|false]" - Impersonate Windows User

userName="Windows user account_to impersonate" | empty string implies impersonate the LOGON user specified by IIS

password="password of above specified account" | empty string

-->

```
<identity impersonate="false" userName="" password=""/>
```

You can optionally specify a fixed user identity other than ASPNET to use in place of ASPNET for all request processing simply by including a valid Windows account user name and password in the userName and password parameters. Each Web application can have its own impersonation identity by supplying a different <identity impersonate> value in each web.config file. When <identity impersonate=true> and userName and password are null ASP.NET will impersonate the effective Windows user identity. For Passport and Forms authentication the impersonated Windows account will always be the IIS anonymous user which by default is IUSR_MachineName. For Windows authentication the impersonated Windows account will be the authenticated Windows user account. Figure 6-4 illustrates the effective Windows user account security context under which ASP.NET requests are processed for each of the possible combinations of authentication and impersonation.

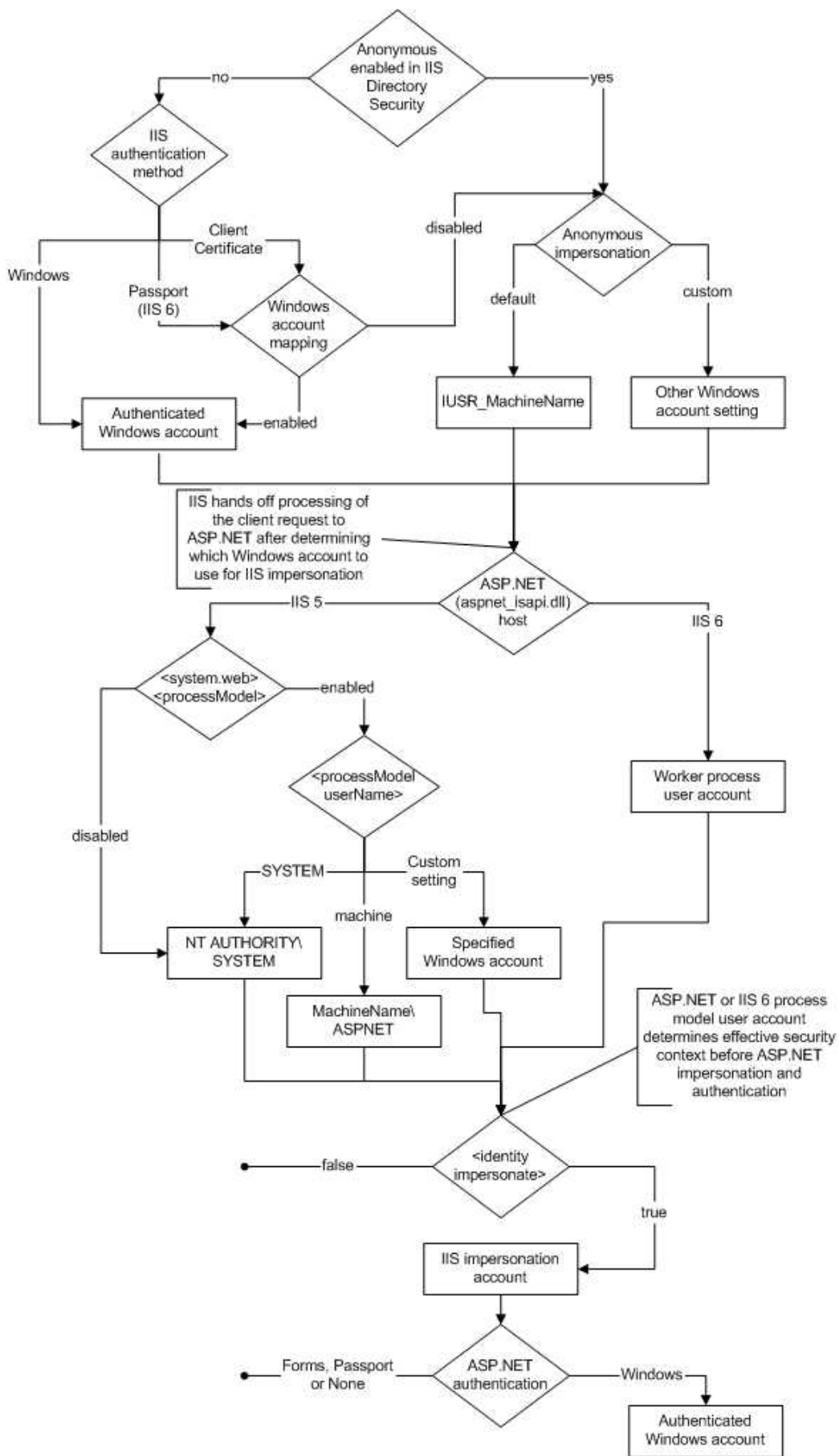


Figure 6-4: IIS and ASP.NET effective user security context impersonation process

ASP.NET impersonation can only impersonate authenticated users when ASP.NET is configured to use Windows authentication. When `<identity impersonate="true">` is set, ASP.NET will switch the effective user security context to the Windows user account supplied by IIS with any ASP.NET authentication method other than Windows. Once an impersonation account is selected, ASP.NET can rely on NTFS file permissions for controlling access to files and normal security permissions in other Windows or network services that support Windows NTLM authentication. No code needs to be written in your ASP.NET application when impersonation is turned on in order to implement a mechanism for access permissions authorization because ASP.NET is able to rely on permissions and authorizations provided by the Windows Server operating system hosting ASP.NET.

Impersonation is only performed for application code. Compilation is always performed using the security context and token of the host process and it results in an assembly that gets written to the Microsoft.NET Framework system subdirectory Temporary ASP.NET Files. Administrative functions performed by ASP.NET are also conducted using the process token including reading XML configuration files and accessing UNC shares, which means that regardless of any impersonation that may occur at the thread-level per-request, host process security context is the one that matters for UNC shares, configuration files, and Temporary ASP.NET Files folder.

Fundamentals of Data Encryption

Encryption is one small part of the larger subject of cryptography which literally means “secret writing”. Encryption is any algorithm used to transform plain text into an encoded representation, cipher text, that can only be decoded through the use of a corresponding decryption algorithm. The encrypted data doesn’t have to be text, of course, any sequence of bits can be encrypted, but the unencrypted data is still referred to as plain text. A key is used in each algorithm, and the keys can either be symmetric (the same in each algorithm) or asymmetric (not the same). The algorithms are together referred to as a cipher, and the key or keys control the way in which the cipher is applied to transform plain text data into cipher text and back again. A cipher is distinguished from a CODEC (enCODer-dECoder), which is an algorithm that just encodes and decodes data to convert it from one form to another, by the use of a key or keys to prevent unauthorized third-parties from decoding (decrypting) cipher text.

Encryption serves two common purposes in ASP.NET: privacy protection and data security. Privacy is provided through SSL encryption applied at the transport layer between Web client and Web server or through application-specific utilization of ciphers.

Data security is accomplished through application-specific utilization of ciphers that prevent unauthorized access to data even when the server’s security policy fails to prevent unauthorized access to storage devices controlled by the server. Privacy is optional; many ASP.NET applications just don’t need it because they don’t accept sensitive data from or deliver sensitive data to the client. Every ASP.NET application needs to use ciphers to provide data security on the server. This section shows you how to use encryption ciphers to enable data security in your ASP.NET application. Chapter 14

shows how to enable SSL encryption between the ASP.NET application and its clients over the network for the purpose of ensuring privacy.

Symmetric ciphers use the same key for both encryption and decryption. The key must be kept secret, and that secret must be shared with both parties: the party that encrypts and the party that decrypts. Protection of the secret is a challenge that never ends, and with automated encryption systems poses a very severe technical problem for which there is no solution: in order to know the secret key so that an automated system can apply it for encryption or decryption the code must have a copy of the secret key accessible at run time. This means that any malicious code that can also run on the box that performs the encryption or decryption can potentially get a copy of the secret and use it for both encryption and decryption. A compromised key is devastating to an automated encryption system that relies on a symmetric cipher.

Asymmetric ciphers are more versatile. Cipher text produced by encrypting plain text data with one of the two keys in an asymmetric key pair can be decrypted only using the decryption algorithm of the asymmetric cipher in conjunction with the other key in the key pair. Consider an ASP.NET application that needs to protect data stored on a server using encryption. The administrator can configure the application to use asymmetric encryption with one key in a key pair and only the administrator, who is in possession of the other key in the key pair, can decrypt the resulting cipher text data. As long as the administrator performs decryption offline using a secure computer, there is very little risk that a malicious third-party will intercept the secret key used for decryption. If a third-party manages to acquire a copy of the encryption key, they may be able to add cipher text to the server's data storage but the third-party can't decrypt data encrypted by the server using the encryption key.

Since the ASP.NET application itself does not know the secret key, all data encrypted by the application is safe from subsequent decryption even if the cipher text later falls into malicious hands along with the key used for encryption. This happens whenever a third party gains control of an ASP.NET application. Asymmetric ciphers are more complex than symmetric ciphers and they generally use larger keys, so they require more computing power to apply. As a result, and due to the fact that a program that has access to plain text has to be somewhat secure to begin with or the plain text would be accessible to third parties prior to being encrypted, symmetric ciphers are used to encrypt large amounts of data and asymmetric ciphers are used to encrypt small amounts of data. A server typically generates a new symmetric key, applies it in a cryptographic transformation, encrypts the symmetric key using an asymmetric cipher, saves the resulting cipher text containing the encrypted symmetric key and the data it was used to encrypt together, and then destroys the symmetric key. This way the relatively small amount of data represented by the symmetric key is protected using the computationally burdensome asymmetric cipher. To decrypt the cipher text containing the symmetric key that was encrypted using one key in an asymmetric key pair requires use of the corresponding key in the key pair which is kept secret by the administrator. Decrypting the cipher text that was produced with the symmetric cipher in order to get back the plain text is possible after decrypting the symmetric key by using the decryption algorithm of the asymmetric cipher.

The cryptographic algorithms provided by the .NET Framework exist in a namespace called System.Security.Cryptography. Within this namespace you will find symmetric and asymmetric cipher classes as well as classes for hashing, digital signatures, and key management. The abstract base classes in this namespace include AsymmetricAlgorithm and SymmetricAlgorithm. An interface, ICryptoTransform, is also defined for use by classes that facilitate cryptographic transformations to encrypt and decrypt data with a particular cryptographic algorithm.

SymmetricAlgorithm classes include DES, RC2, Rijndael, and TripleDES. Each of these classes is abstract with a separate derived CryptoServiceProvider class and a static member method called Create that creates a derived class instance. DES has the class DESCryptoServiceProvider, RC2 has RC2CryptoServiceProvider, Rijndael has RijndaelManaged, and TripleDES has TripleDESCryptoServiceProvider. The derived class is not abstract, so it's the one you use directly from your code to perform plain text-to-cipher text and cipher text-to-plain text transformations. The SymmetricAlgorithm base class includes a method named CreateEncryptor that returns an object that implements the ICryptoTransform interface. For the symmetric ciphers supported by the .NET Framework other than Rijndael the ICryptoTransform object returned by CreateEncryptor can be type cast to (CryptoAPITransform), the simplest of ICryptoTransform implementations. Rijndael encryptors must be type cast to (ICryptoTransform) so it is usually better to use the interface in your code rather than CryptoAPITransform unless you have a good reason to do otherwise. These SymmetricAlgorithm classes each rely on a symmetric key, which they will generate for you automatically if you don't provide one. They perform transformations of plain text one block at a time, with the block size determined by the cipher settings.

Encrypting Data

SymmetricAlgorithm classes also accept an optional initialization vector, a second shared secret that is used as a random seed to introduce noise into the cipher text so that patterns do not emerge based on patterns in the plain text that can dramatically simplify decryption by a malicious third-party. For a given block of plain text data, an encryption algorithm and a given key will always result in the same cipher text output. For this reason, the initialization vector is used to introduce initial randomness and feedback is added where the result of previous block encryption impacts encryption of subsequent blocks. System.Security.Cryptography includes a CipherMode enumeration shown in Table 6-1 that controls whether the initialization vector is used and how feedback is added. Consider the following two-part example.

```
String s = "plaintext";  
byte[] plaintext = new byte[Encoding.ASCII.GetByteCount(s)];  
int bytes = Encoding.ASCII.GetBytes(s,0,s.Length,plaintext,0);  
Response.Write(BitConverter.ToString(plaintext));
```

The code shown produces the following output representing the hexadecimal encoded value of each character in the input string "plaintext" separated by dashes. Note that the hexadecimal value for each letter matches the ASCII table; the code shown does not encrypt the ASCII characters, it simply encodes the plain text as ASCII.

70-6C-61-69-6E-74-65-78-74

When the RC2 cipher is applied with its default settings to encrypt the string “plaintext” using a null key, that is, a key of the default bit-length for RC2 (128 bits) where each bit is zero, in hex a value of 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00, using an initialization vector that is also null and is equal in length to the default RC2 block size of 64 bits, the cipher text output is always 3C-0A-AD-AC-96-C6-F1-4F-3E-89-BC-56-07-3B-A5-04. The following code demonstrates this by generating a null key and a null initialization vector sized according to RC2 defaults.

```
String s = "plaintext";
byte[] plaintext = new byte[Encoding.ASCII.GetByteCount(s)];
int bytes = Encoding.ASCII.GetBytes(s,0,s.Length,plaintext,0);
byte[] ciphertext;
int a;
ICryptoTransform Encryptor;
SymmetricAlgorithm RC2Crypto = RC2.Create();
System.IO.MemoryStream msKey = new System.IO.MemoryStream(RC2Crypto.KeySize/8);
for(a = 0;a < RC2Crypto.KeySize;a = a + 8)
{ msKey.WriteByte(Byte.MinValue); }
RC2Crypto.Key = msKey.ToArray();
System.IO.MemoryStream msIV = new System.IO.MemoryStream(RC2Crypto.BlockSize/8);
for(a = 0;a < RC2Crypto.BlockSize;a = a + 8)
{ msIV.WriteByte(Byte.MinValue); }
RC2Crypto.IV = msIV.ToArray();
Encryptor = (ICryptoTransform)RC2Crypto.CreateEncryptor();
ciphertext = Encryptor.TransformFinalBlock(plaintext,0,bytes);
Response.Write(BitConverter.ToString(ciphertext));
```

It's important to work with keys and initialization vectors as byte arrays rather than as integers or strings. For one thing, 64 bits is the largest integer type available in the .NET Framework and encryption keys often exceed 64 bits. For another thing, you can derive byte arrays from strings but the opposite is not always true. Use System.BitConverter, System.IO.MemoryStream or other classes that enable the manipulation of byte arrays to work with these data elements.

The example does not set CipherMode explicitly, so the default mode is used which is Cipher Block Chaining (CBC) as for all SymmetricAlgorithm classes. CBC is a cipher mode that uses the initialization vector to supplement the encryption and decryption algorithms of the underlying cipher. To decrypt cipher text created using a cipher in CBC mode the decryptor must have both the key and the initialization vector and also apply the cipher's decryption algorithm in CBC mode.

Table 6-1: System.Cryptography.CipherMode Enumeration Values

Enumeration Initialization Vector and Feedback Properties

CBC Cipher Block Chaining mode takes the cipher text that results from encryption of the previous block and uses it to encrypt the next plain text block using an exclusive OR bitwise operation before the SymmetricAlgorithm is applied. An initialization vector is

used in place of cipher text the first time the exclusive OR bitwise operation is performed in order to add noise to the resulting cipher text.

CFB Cipher Feedback mode uses the initialization vector or the previous cipher text block to perform a bitwise shift operation prior to applying the SymmetricAlgorithm one byte at a time. The cipher's feedback size must be set to 8 bits in order for this CipherMode to be used.

CTS Cipher Text Stealing mode is similar to CBC mode but produces output that is identical in length to the input plain text.

ECB Electronic Codebook mode does not use the initialization vector nor any feedback mechanism. This mode encrypts each block independently as dictated by the raw underlying SymmetricAlgorithm.

OFB Output Feedback mode is similar to CFB mode but uses a different procedure for determining feedback used in the bitwise shift operation.

To understand the difference between the CBC cipher mode that uses an initialization vector and one that does not, you need only switch from CBC mode to Electronic Codebook (ECB) mode and repeat the encryption. The Mode property of the SymmetricAlgorithm base class enables the CipherMode to be set. With ECB mode set prior to the call to CreateEncryptor in the example shown, the cipher text is:

3C-0A-AD-AC-96-C6-F1-4F-6A-44-69-03-6C-17-04-CC

You'll notice if you compare the CBC mode output with the ECB mode output that the first 8 bytes of cipher text are the same, with the next 8 bytes changing to 3E-89-BC-56-07-3B-A5-04 when ECB mode is used. This is due to the fact that RC2 uses a 64-bit (8 byte) block size. In CBC mode the result of the first block encryption with a null initialization vector always matches the first block encryption in ECB mode which uses neither the initialization vector nor the previous cipher text block to perform any exclusive OR operations on subsequent plain text blocks. ECB applies the raw cipher without feedback to each plain text block, and therefore the first block of CBC CipherMode generated cipher text produced without cipher text feedback happens to match the first block of ECB CipherMode generated cipher text. Table 6-2 lists the CipherModes supported by each SymmetricAlgorithm.

Table 6-2: SymmetricAlgorithm CipherModes

	CBC	CFB	CTS	ECB	OFB	
DES	DEFAULT		No Support	No Support	Supported	No Support
RC2	DEFAULT		Supported	No Support	Supported	No Support
Rijndael		DEFAULT		No Support	No Support	Supported No Support
TripleDES		DEFAULT		Supported	No Support	Supported No Support

The amount of feedback introduced into subsequent encryption blocks when CBC mode is used is variable. CFB mode uses a fixed 8-bit feedback size. The feedback size is set using the FeedbackSize property of the SymmetricAlgorithm derived class. In no case can FeedbackSize exceed the BlockSize setting. Valid block and key sizes for each SymmetricAlgorithm cipher and the default feedback size used in each are shown in Table 6-3.

Table 6-3: Key, Block, and Default Feedback Sizes for Symmetric Algorithms

	Valid Key Sizes	Valid Block Sizes	Default Feedback Size
DES	64 (default)	64 (default)	8 bits
RC2	40, 48, 56, 64, 72, 80, 88, 96, 104, 112, 120, 128 (default)	64 (default)	8 bits
Rijndael	128, 192, 256 (default)	128 (default), 192, 256	128 bits
TripleDES	128, 192 (default)	64 (default)	8 bits

Each of the SymmetricAlgorithm classes in each supported CipherMode will automatically generate encryption key and initialization vector values of the default sizes listed in Table 6-3. The default size of an initialization vector is always equal to the default block size. In any case, when using any CipherMode other than ECB you must keep a copy of the initialization vector as well as the key used in the encryption in order to decrypt the resulting cipher text. If you change the default feedback size, the decryptor must also be configured to use the same number of feedback bits or decryption will fail. Symmetric ciphers rely on shared secrets for data security, including initialization vector and cipher settings in addition to key.

Decrypting Data

To decrypt cipher text you need to know four things: the cipher used, the cipher's settings including number of feedback bits and the CipherMode setting, the secret key used to encrypt the plain text, and the cipher's initialization vector, if any. As long as your code provides each of these items correctly to the decryptor that is instructed to transform the cipher text back into plain text, the transformation will succeed. The following code shows how the cipher text produced in the previous example is decrypted at a later time by different code.

```
int a;
ICryptoTransform Decryptor;
SymmetricAlgorithm RC2Crypto = RC2.Create();
System.IO.MemoryStream msDecryptionKey = new
    System.IO.MemoryStream(RC2Crypto.KeySize/8);
for(a = 0;a < RC2Crypto.KeySize;a = a + 8)
{ msDecryptionKey.WriteByte(Byte.MinValue); }
RC2Crypto.Key = msDecryptionKey.ToArray();
System.IO.MemoryStream msDecryptionIV = new
    System.IO.MemoryStream(RC2Crypto.BlockSize/8);
for(a = 0;a < RC2Crypto.BlockSize;a = a + 8)
{ msDecryptionIV.WriteByte(Byte.MinValue); }
RC2Crypto.IV = msDecryptionIV.ToArray();
byte[] ciphertext;
ciphertext = [read from cipher text storage]
Decryptor = (ICryptoTransform)RC2Crypto.CreateDecryptor();
byte[] plaintext;
plaintext = Decryptor.TransformFinalBlock(ciphertext,0,ciphertext.Length);
Response.Write(BitConverter.ToString(plaintext));
Response.Write("<br>");
Response.Write(Encoding.ASCII.GetString(plaintext));
```

The same null key and initialization vector byte arrays are created by this decryption code as were used in the original encryption step, and the RC2 cipher is likewise applied with its default settings. The code shown does not specify a particular storage mechanism and reading cipher text bytes into the byte[] array from a storage location is application-specific. Cipher text is passed to the Decryptor object in a call to its TransformFinalBlock and the resulting plain text bytes are stored in a byte[] array named plaintext. A BitConverter object is used to write plain text bytes:

70-6C-61-69-6E-74-65-78-74

To simplify the common task of reading and writing cipher text data from and to files and network storage, System.Security.Cryptography includes a CryptoStream class that derives from System.IO.Stream. Use CryptoStream exactly as you would use the network stream but wrap it in a CryptoStream and you get encryption and decryption automatically. The CryptoStream constructor accepts a Stream object and an object that implements the ICryptoTransform interface. Any stream, including file streams, can be turned into a CryptoStream and a cipher applied. Simply construct an instance of the cipher class and call CreateEncryptor or CreateDecryptor to obtain the ICryptoTransform object to pass to the CryptoStream constructor based on whether you want encryption or decryption.

The sample shown here uses RC2 with a null key in order to demonstrate encryption and decryption using a key that is simple for demonstration purposes. However, such a key is cryptographically unsafe due to the fact that it does not introduce enough variability in the resulting cipher text. RC2 lets you use a weak key, but Rijndael does not, it will throw an exception when you attempt to use a key that is known to be cryptographically weak. Key generation is one of the essential elements of conducting encryption in real applications. Key generation is almost as important as key management and these topics are explained in the next section.

Generating and Managing Encryption Keys

SymmetricAlgorithm classes will automatically generate encryption key and initialization vector values if your code doesn't provide them prior to attempting an encryption operation. You can call SymmetricAlgorithm.GenerateKey or GenerateIV explicitly or let the SymmetricAlgorithm derived class do it for you. The only thing you have to remember to do is save a copy of these values after producing cipher text otherwise it will be difficult to decrypt later. The Key and IV properties provide access to these values from within your code. To specify the Key and IV values explicitly you simply set them prior to calling CreateEncryptor or CreateDecryptor as shown in the previous examples.

RandomNumberGenerator is a useful class for quickly and easily creating random keys of specific bit lengths for symmetric ciphers. The following code generates a random 128-bit key using RandomNumberGenerator.

```
byte[] randomkey = new byte[16];
RandomNumberGenerator.Create().GetBytes(randomkey);
Response.Write(BitConverter.ToString(randomkey));
```

For AsymmetricAlgorithms you generate a pair of related keys rather than a single key.

The algorithm used to generate a key pair is intimately intertwined with the workings of the cipher, so your only option for generating key pairs is to use a utility provided especially for use with the AsymmetricAlgorithm. As with SymmetricAlgorithms, simply creating a new instance of the class causes keys to be generated for you automatically. The following code generates a key pair and uses the public key to encrypt plain text then outputs the key pair formatted as XML.

```
String s = "plaintext";
byte[] plaintext = new byte[Encoding.ASCII.GetByteCount(s)];
Encoding.ASCII.GetBytes(s,0,s.Length,plaintext,0);
RSACryptoServiceProvider RSACrypto = new RSACryptoServiceProvider();
byte[] ciphertext = RSACrypto.Encrypt(plaintext,false);
Response.Write(RSACrypto.ToXmlString(true));
```

This code will only work when run under SYSTEM security context due to default restrictions on use of RSA algorithms from within ASP.NET. It produces output like the following:

```
<RSAKeyValue><Modulus>wbEEM6Xf87Hfwh/TO9Rd5yTKRYB/ipRgpbzwKwsiDw+JgbuXzy
7i5ohHzA5s7iMq22LQigbyGJDvEfWRh7p5k5k3/qAsq5XW1CWkxhVYEjPYT2aus1Kwcp
AqVk4DmngTwl8MaTFghph2iC+LzDq28kiPyzlrPFmdqKP7lgLeZEK=</Modulus><Expon
ent>AQAB</Exponent><P>+ACpWzZYwBpJ1T03876uu4jxcU+xbYQRqvDmBeHN41Sb6
FRPKMUad0BJqS6kaRvsYc5q/zTGJgPGm15136fznw==</P><Q>x+//7myOFGXf8RQmy
ek0Ysnfcl8TtUPZfcdxEfSTxq/lgyZ0jDYw+iOZOpBJYArDbav834bMyGQgQzXx+HDfFw=
=</Q><DP>4qA4lpHXKDTdo2794k8tfVH20ITyrhEx0/OvP1DIxCRdFEF21NrJBjBkV79Pn
n1V1Uq7m9qt968bnn8DWA4yIQ==</DP><DQ>utpRFUHehrGu2F884PZRPwHrEbhJct4
2JKU39s/cS5N8kRUfVupOW3dpfJHcASYN/jD94ujX+W+ZtzZzLPxPgQ==</DQ><Inverse
Q>WnZtq914IOZ9u6rL2/CjYuwaAJMooWypP1NmZHUdAPTA74ChdGkxM3iuU8ua5Z
PEWjgqu+m45v3FdNtk5wuA==</InverseQ><D>O32EHdznsTDD2hrRSUQBmuNWNW
D1uuF18H1PjM4LcoG4PreQLtU45ud+bXAjU/t3N430P0bJKJ3W1vCbB7BMiO7PiHLRMx
hMwj+U/Jh4cRjCtCjJn4+fEARs+CO+TCJfCzE2zc4sRYPVvUFP1k6pEVGhWRWkaU3Ng
AuCinKGphU=</D></RSAKeyValue>
```

To load a public key only so that your code can decrypt cipher text produced using a private key with an AsymmetricAlgorithm like the RSA cipher you import the public key into an instance of the corresponding CryptoServiceProvider class. If you previously exported an asymmetric key pair to XML using ToXmlString method then you can use just the public key node <Modulus> and the <Exponent> to feed the public key from the key pair back into the CryptoServiceProvider in order to decrypt. The ToXmlString method of the AsymmetricAlgorithm base class includes a Boolean parameter indicating whether or not to output the private key in addition to the public. Pass in a false value in order to output only <Modulus> and <Exponent> XML nodes. Call the FromXmlString method and supply it with a String parameter containing the <Modulus> and <Exponent> XML nodes only, which represent a public key.

```
String publickey = [ <RSAKeyValue>XML</RSAKeyValue> ]
RSACrypto.FromXmlString(publickey);
```


It's important to understand that any time a key is stored in memory, even temporarily, it is vulnerable. Malicious code, if it can read the memory in which the key is stored, can intercept the key. There is only one way to avoid this vulnerability: never generate keys on vulnerable network-connected computers and never use keys to perform encryption or decryption on vulnerable network-connected computers. These measures aren't very practical, of course, since ASP.NET applications run on network-connected computers that are relatively vulnerable compared to unconnected computers. The solution is not to use symmetric ciphers to encrypt data that is critically important. Use asymmetric ciphers instead, so that the key used to encrypt data is irrelevant to decryption of the cipher text. Generate asymmetric key pairs offline and keep one of the keys completely hidden and inaccessible from any network. Transport cipher text data to a dedicated decryption computer where only an authorized user can perform decryption transformations.

Verifying Data Integrity Using Hash Codes

One of the things you tend to worry about when you've deployed code to the Internet as part of your Web application is that a malicious third party will change the code somehow. Ideally your server security is impenetrable so that you can sleep well at night and let your system administrator worry about such things. But it's always best to plan for the worst and put in place simple and manageable safeguards. If your ASP.NET application could automatically detect unauthorized changes and lock down access to modified files, one of the worst-case scenarios for ASP.NET security could be mitigated effectively.

`System.Cryptography` includes classes that implement hash algorithms. A hash is a short binary sequence of fixed length that is computed based on an input of arbitrary length. The hash algorithm is designed to make it statistically improbable for a malicious third party to discover additional inputs to the hash algorithm that will compute the same output hash as another input. Any change to the input creates substantial unpredictable changes to the hash value output by the algorithm. The original data used as the input to the hash algorithm can't be reverse engineered based on the hash value. A common use of hash values is as a means to detect whether or not changes have been made to arbitrary data since the last time the hash value was computed. This makes hash algorithms ideal for use in detecting unauthorized changes to your ASP.NET application files. The trick is layering-in the validation of hash values whenever one of your ASP.NET application files is accessed by a client.

ASP.NET provides a mechanism for layering-in precisely the functionality required to compute and validate hash values for the source of every page requested by a client. The previous discussion in this chapter included descriptions of the `global.asax` events for authentication and authorization, and those events are suitable for layering-in hash code validation as well. However, if your goal is to add a security failsafe that will afford more protection to your ASP.NET application in the event of a malicious third party obtaining authoring permissions on your server, you can be pretty certain the unauthorized intruder will end up with the ability to edit `global.asax` and in so doing remove any failsafe code you may have placed there. A better solution is to layer-in this type of functionality at a level that even an authorized ASP.NET application author is unable to change. For example, by editing `machine.config`. The following code demonstrates how to create a new HTTP module class that you can register within `<httpModules>` of `<system.web>` in `machine.config`.

```

using System;
using System.Web;
using System.Security.Cryptography;
using System.IO;
namespace HashVerification {
public class HashVerificationModule : System.Web.IHttpModule {
public void Init(HttpContext context)
{ context.AuthorizeRequest += new EventHandler(this.HashAuthorization); }
public void Dispose() {}
public void HashAuthorization(object sender, EventArgs e) {
HttpApplication app = (HttpApplication)sender;
try{FileStream f =
File.Open(app.Request.PhysicalPath, FileMode.Open, FileAccess.Read,
FileShare.ReadWrite);
HashAlgorithm md5Hasher = MD5.Create();
byte[] hash = md5Hasher.ComputeHash(f);
f.Close();
if(!BitConverter.ToString(hash).Equals(
"8C-6A-4D-C3-1B-05-F8-20-81-A5-5A-3A-C6-12-6B-C8"))
{ throw(new Exception()); }
}
catch(Exception ex) {
app.Response.Write(
"<html><body><h1>Error Processing Request</h1></body></html>");
app.CompleteRequest(); }
}}

```

System.Web.IHttpModule is the interface that all HTTP modules implement. The default HTTP modules that were discussed previously in this chapter, such as the WindowsAuthenticationModule and UrlAuthorizationModule, each implement this interface. The idea behind an HTTP module is that it gets activated by ASP.NET before any requests are processed so that it can register itself as an event handler for whatever events it needs to intercept in order to layer-in new functionality. In the past you would have built an ISAPI Filter DLL for this purpose. HTTP modules are superior to ISAPI Filters for a variety of reasons including that they are built as managed code and therefore are not susceptible to buffer overflow attacks and they can be loaded and unloaded without restarting IIS.

The IHttpModule class shown in the example registers itself with the Application's AuthorizeRequest event. It adds an authorization requirement that validates the hash value for the ASP.NET page being requested by the client matches the hash value as it was computed previously. It does this by opening a FileStream for the Request.PhysicalPath and using the MD5 HashAlgorithm class to compute a hash from the file input stream. The following lines of code do this, where f is the FileStream returned by the File.Open call:

```

HashAlgorithm md5Hasher = MD5.Create();
byte[] hash = md5Hasher.ComputeHash(f);

```

The example code then does something that your real hash validation HTTP module will not do, it compares the hash value so computed with a hard-coded hash value as computed previously for the single page that was used for testing purposes. To turn this example into a real-world module, you simply need to code a mechanism by which the previously computed hash value is retrieved from a secure storage such as a database keyed on the Request.PhysicalPath. Whenever you deploy updates to your ASP.NET application source files, simply update the database of stored hash values to let your hash validation HTTP module know the new hash values. To Configure the hash validation HTTP module in machine.config you need to assign its assembly a strong name and modify <httpModules> by adding a new <add /> line like the following. Add the class and its assembly to the global assembly cache first.

```
<add name="HashVerification" type="HashVerification.HashVerificationModule, assembly"/>
```

A hash can also be cryptographically signed using an AsymmetricAlgorithm class, RSA or DSA. RSACryptoServiceProvider and DSACryptoServiceProvider, the RSA- and DSA-derived classes respectively, contain a SignHash method that computes a hash and applies a private key to encrypt the hash value. The corresponding VerifyHash uses the public key from the key pair to verify the signature. For absolute confirmation that your ASP.NET application files have not been tampered with, you can apply a signature to the hash values you store in your hash database and call VerifyHash prior to performing the hash validation so that a malicious third-party would be required to steal your secret key and compromise both your Web authoring security and your database security in order to bypass the hash verification HTTP module loaded into each ASP.NET application by machine.config.

Chapter 7: Secure Scripting for Web Applications

This chapter is primarily about secure Active Server Pages scripting, but it should also be read by ASP.NET developers and administrators who manage ASP.NET deployments because Web application security fundamentals are emphasized in this chapter whereas the previous chapter, Security in ASP.NET, focused on the security features and useful threat countermeasures specific to Web applications created using the Microsoft .NET Framework. Most of this chapter will be useful to any developer or administrator of Web applications regardless of the server-side script engine being used even though the code samples are all written in classic ASP. This chapter explains critical, pervasive security flaws such as Cross Site Scripting (XSS) that impact every Web application hosting and development platform not just ASP and IIS. The design of the typical Web browser with its various enhanced capabilities, such as client-side scripting, create serious problems for Web application security.

Many Web applications incorrectly assume that the network client will always be a conventional Web browser under the interactive control of a human user and fail to consider malicious automated programs as potential clients, which causes additional problems for security. There are a number of solutions to these various problems, and this chapter explains a couple of them, but unfortunately there are many bad solutions in use today that just make problems worse or create new problems even if they do resolve old ones. Certain Microsoft products built around IIS have features that fall into this latter category. For example, Site Server and Site Server Commerce Edition provide a session management and client authentication feature known as automatic cookie authentication. This feature, and any like it, must never be used and should never have been deployed in the first place because of fundamental security problems with the architecture of the Internet.

When vendors or developers of custom application logic fail to account for the architectural and security realities of the Internet, bad code results and security problems are created where none should have existed in the first place.

The goal of secure Web application development and server-side scripting is to avoid creating new vulnerabilities and properly harden Web applications against all known threats.

Know Your Audience/Enemy

The potential audience of any Web application includes the typical human user wielding a Web browser but it also includes malicious client code that executes automatically from a number of possible locations. For instance, a malicious script might be deployed to a Web server by an attacker, worm, or virus that causes the Web server to act as a client of its own application services. Web browsers can be sent malicious JavaScript that causes the browser to do the bidding of the malicious script author without the knowledge or consent of a human computer operator. Even intranet Web applications must take these threats into consideration because e-mail messages sent to intranet users can potentially contain code inline or as attachments. Malicious content delivered through Web browsing

can compromise or hijack intranet client nodes and cause them to attack an intranet Web application. Then there are the internal threats from third parties who gain access to the intranet network. And don't forget rogue employees. The first few lines of any ASP script should enforce blunt security policy such as denying access to local address ranges. For example:

```
<% Response.Buffer = TRUE
addr = Request.ServerVariables("REMOTE_ADDR")
If Left(addr,3) = "127" or Left(addr,7) = "192.168" or Left(addr,3) = "10." or Left(addr,7) =
"169.254" Then
    Response.Clear
    Response.End
Else If Left(addr,3) = "172" Then
    addr = Mid(addr,5)
    addr = Left(addr,InStr(addr, "."))
    If CInt(addr) >= 16 and CInt(addr) <= 31 Then
        Response.Clear
        Response.End
    End if
End if %>
```

This assumes, of course, that the ASP script is only meant to be used by clients located on the Internet and that your network doesn't employ a reverse proxy that rewrites source address information thereby preventing IIS from perceiving any source IP address other than the one assigned to the reverse proxy. Denying access to local address ranges, in particular addresses such as 127.0.0.1 that would potentially appear in the source address of a request sent by the server to itself, is important even if you also have TCP/IP filtering enabled. ASP scripts deployed as part of intranet applications should likewise enforce an appropriate blunt security policy like refusing to service requests from any address outside the range used by intranet network nodes. It's tempting to leave intranet Web applications unhardened or rely on automatic network logins using Windows authentication through Lan Manager-based Windows NT domain accounts or Kerberos-based Active Directory authentication for Windows 2000/XP and .NET Server. But automatic login to password-protected network services based on Windows networking leaves everything open to attack the moment unfriendly code gains access to a user security context that is automatically trusted to access these services.

Securing Anonymous Session State

The security of information contained in session state is one of the primary issues facing Web applications. Access to session state information and the acceptance of data for storage into it must be conducted in accordance with the same security measures that are explained in the remainder of this Chapter. In addition, authenticated sessions must conform to proper security procedures for Web-based authentication as described in Chapter 12. As long as the proper countermeasures are in place, sensitive data in session state will be no more vulnerable than information stored in a database or otherwise accessible to IIS on the server-side. There is, however, one special consideration that is unique to sessions and must be accounted for separately:

anonymous sessions that include session state data. When you know that your application allows anonymous sessions, special care must be taken.

Anonymous sessions are more difficult to protect properly than are authenticated sessions because of Internet architecture realities not the fact that the end user is anonymous. There's nothing inherently insecure about receiving requests from anonymous users' Web browsers. The threat lies in the way that your application greets a new anonymous user, establishes session state storage, assigns a session identifier to the anonymous session and transmits the session identifier to the client so that the identifier can be relayed back to IIS in each subsequent request. When SSL encryption is not used to protect the contents of HTTP responses from interception in-transit, those cleartext responses are subject to caching, including Set-Cookie: HTTP headers. When a cookie is dropped in response to a GET request for a URL that does not include QueryString name/value pair parameters, proxy servers and other entities responsible for cache management are prone to consider any subsequent request for that same URL to represent a cache hit and serve the previous response out of cache inclusive of the previous Set-Cookie header. There is more on cache-busting later in this Chapter, but cache-busting techniques don't resolve the entire problem of securing anonymous sessions.

Authenticated sessions are relatively easy to harden against cache threats because the authentication step will never be perceived as a cache hit. This is because authentication credentials are by definition unique to the user who possesses them. No two users will supply the same authentication credentials, and any network device that manages cache must allow these credentials to pass to the server and must not ignore them when determining whether or not the request can be serviced out of cache. However, prior to the authentication step where credentials are transmitted, any communication received from the user's Web browser falls into the category of anonymous traffic, creating an initial anonymous session, potentially. The anonymous session must be converted to an authenticated session after authentication occurs, and the way in which this happens is subject to security vulnerabilities and caching as well. If your anonymous sessions and the conversion to authenticated sessions are not properly secured, then it's possible for multiple users to end up sharing the same authenticated session, the impact of which is likely to be bad. This results in both authenticated users sharing the authenticated session state that belongs to the user who authenticates last. This scenario is just one undesirable result that may occur if session state is converted from anonymous to authenticated by way of a change to session state on the server-side only instead of both a server-side state change and a change to the session identifier used to associate the client with a particular session.

Meeting and Greeting Anonymous Users

The most common way to associate a client with a particular session ID is through the use of cookies. Unlike authentication credentials, cache management does not always consider the presence of a cookie as a factor that distinguishes one request from another for the purpose of identifying cache hits. This is one reason that cookies should never be mistaken for authentication.

For more on known problems with cookies see Microsoft Security Bulletin MS99-035 <http://www.microsoft.com/technet/security/bulletin/ms99-035.asp>

Cookies can be used for flawless client session tracking, but only when used in conjunction with SSL to prevent cache storage of Set-Cookie headers in HTTP responses and inappropriate cache hits by cache managers that ignore the presence of a Cookie header in HTTP requests. For almost-flawless operation, you must carefully avoid issuing Set-Cookie headers in response to HTTP requests that use the GET request method. You can't be absolutely certain that IIS will be hit when an anonymous user browses Web pages without SSL encryption if the user's Web browser sends GET requests because cached responses sent to past users might be delivered to the browser instead, even when the HTTP responses contain Set-Cookie headers. If caching were to stop completely on the network simply because cookies are used, content caching would virtually disappear.

Another technique commonly used when cookies are disabled in the client browser is to URL-encode a session ID through dynamic modification of the hyperlink URLs embedded in HTML content. An initial HTTP Redirect (302 result) is sometimes used as the first HTTP response sent to the client. The logic for this approach is simple, if there is no session ID value encoded in the URL of the request, don't service it. Instead, use Response.Redirect to send the browser to a different address that includes a dynamically assigned session ID encoded in the URL as a name/value pair. However, this technique is even more prone to inappropriate cache hits based on the reality that different users may enter a site through different initial URLs, not everyone comes in the front door using a URL that has no QueryString parameters. As other people attempt to hyperlink to a site that URL-encodes session ID values, they are also prone to mistaking the QueryString parameters as a necessary part of the URL of the page to which they wish to link. This causes each user who clicks on such a hyperlink that contains a hard-coded session ID to share the same session state, even if SSL is used, as long as the hard-coded session ID remains valid.

Figure 7-1 shows one real-world occurrence of a hard-coded session ID embedded in a hyperlink from one site to another. The result was shared anonymous session state such that as anonymous users filled out the HTML FORM to register to receive a free copy of the Microsoft Press Book "Business Intelligence: Solutions for Making Better Decisions, Faster" as well as download a free Power Pack and enroll in a home theater system giveaway, the next anonymous user who clicked on the same hyperlink would see the contact information entered into the HTML FORM by the previous visitor. In the figure shown, you can see firstname and lastname fields at the bottom both contain values pre-populated out of anonymous session state.

```

Microsoft Solutions for Business Intelligence Register.htm - Notepad
File Edit Format Help
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!-- saved from url=(0084)http://www.powerofbusinessintelligence.com/register.asp?
uid=12123,37487.321884567900 -->
<HTML><HEAD><TITLE>Microsoft Solutions for Business Intelligence: Register</TITLE>
<META http-equiv=Content-Type content="text/html; charset=iso-8859-1"><LINK
href="Microsoft Solutions for Business Intelligence Register_files/style.css"
type=text/css rel=stylesheet></HEAD>
<P class=Base>To request your Power Pack, download your Library of
business intelligence information and enter for a chance to win a
Panasonic SC-HT70 DVD Home Theater System, please verify or update
the following information and click <B>Submit</B>. </P>
<TABLE cellspacing=0 cellpadding=6 width=527 border=0>
<TBODY><TR valign=top><TD width=30><INPUT type=checkbox value=1 name=rg_book></TD>
<TD class=Base width="100%"><STRONG>Yes!</STRONG> I'd like a
copy of the Microsoft Press Book <I>Business Intelligence:
Solutions for Making Better Decisions, Faster</I> (U.S.
residents only).</TD></TR></TBODY></TABLE>
<TABLE cellspacing=0 cellpadding=6 width=527 border=0>
<TBODY><TR valign=top>
<TD width=30><INPUT type=checkbox value=1 name=rg_contest></TD>
<TD class=Base width="100%"><B>I want a chance to win!</B>
Enter me in the sweepstakes for the Panasonic DVD Home Theater
System (<A href="http://www.powerofbusinessintelligence.com/rules.asp"
target=pop>sweepstakes rules</A>).</TD></TR></TBODY></TABLE>
<TABLE cellspacing=0 cellpadding=6 width=527 border=0>
<TBODY><TR valign=top><TD width=30><INPUT type=checkbox CHECKED value=1
name=rg_powerpack></TD><TD class=Base width="100%"><B>Yes!</B> I'd like a copy of
the Microsoft/Comshare Business Intelligence Power Pack.
</TD></TR></TBODY></TABLE><!-- BEGIN contact info -->
<P><IMG height=11 src="Microsoft Solutions for Business Intelligence
Register_files/grDOTDIV.gif" width=527></P>
<P class=Subhead>Contact Information</P>
<TABLE cellspacing=0 cellpadding=6 width=527 border=0>
<TBODY><TR valign=top><TD class=Base colspan=2>Required fields are <SPAN
class=Question>bold</SPAN> and <SPAN class=Question>color</SPAN>.</TD></TR>
<TR valign=top><TD class=Question width=150>First Name</TD>
<TD width=377><INPUT maxLength=50 size=30 value=Egon name=firstname></TD></TR>
<TR valign=top><TD class=Question>Last Name</TD><TD>
<INPUT maxLength=50 size=30 value=Streibah name=lastname></TD></TR>

```

Figure 7-1: Anonymous Session State Revealed to Other Visitors with the Same Session ID

This was probably done by the Web application developer to make it easier for users to correct data entry mistakes when a round trip with the server is required. By populating the HTML FORM fields with whatever values are stored in the anonymous session state, the developer reasoned that the end user would be saved time as they would only have to enter values for fields that are blank and entering values for all fields over again would be unnecessary. Unfortunately, the Web application was designed to serve content out of session state even when no error condition occurred, and without consideration of the possibility that the session ID provided by the end user's browser may not in fact belong to that user's anonymous session. This example demonstrates the importance of careful handling of first contact with a new anonymous user. A session ID provided by a browser in its very first request to the server should always be rejected, even if it does match a valid session ID active in session state currently, because the browser isn't supposed to tell the server what session ID it wishes to use, the server is supposed to assign a session ID to a particular browser. This is a simple security policy that must be enforced. Code like the following can be used to detect first contact with an anonymous client.


```

<% sSessionID = Request.Cookies("sessionid")
If sSessionID = "" Then
    sSessionID = Request.QueryString("sessionid")
End if
If sSessionID <> "" Then
    sReferrer = Request.ServerVariables("HTTP_REFERER")
    sHost = Request.ServerVariables("HTTP_HOST")
    If sHost <> "" and sReferrer <> "" Then
        If InStr(1,sReferrer,sHost,1) = 0 Then
            sSessionID = ""
        End if
    End if
End if
End if %>

```

The ASP script shown implements a simple REFERER validation to make sure any request that provides a sessionID does not also provide a foreign address in the HTTP REFERER header. When the sSessionID variable is empty after this script block is finished, the application knows that the request is the first one received from an anonymous client and a new session ID and anonymous session state need to be established. A foreign REFERER is a sure sign that the session ID supplied should be ignored because it means that the user clicked a hyperlink on a different site to arrive at the present URL where the ASP script lives. Note that if you disallow URL-encoded session ID and force all of your Web site users to accept cookies as a condition for anonymous access to your Web site, then you won't have to worry about foreign REFERER. This is a privacy policy and usability design decision that you'll have to make on a case-by-case basis. For many Web sites it is unacceptable to turn away users who refuse to allow cookies, and URL-encoding the session ID is your only option for session state tracking in these instances.

Based on the determination made by the code as shown that this is the first time servicing the particular client, it's easy to see that the session ID offered by the client must be ignored. Such foreign sources of session ID violate the security policy that says the server is the only authority that can assign session identifiers to clients. Persistent sessions are an unacceptable security risk. All sessions must expire after a short amount of inactivity, such as 20 minutes. However, your application can still get the usage tracking and user profiling details it desires by keeping track of all sessions assigned in series to the same client browser. The session ID offered by the browser may refer to an expired session, but that doesn't mean your server has discarded all knowledge of who the user was and what the user did during that session. Keep whatever information your server needs for usage profiling, but be sure not to reveal any of it to the anonymous user. There is always a chance with an anonymous session that the end user who is in control of the browser is not the same person as last time. This is particularly true on shared computers. You must avoid making any assumptions in your application about a single computer equating to a single user.

Avoiding anonymous session state entirely, so that all anonymous users have the same session state: none; or requiring SSL as part of the process of meeting and greeting a new anonymous client who does not yet possess an anonymous session ID are the only guaranteed-secure ways to handle anonymous visitors. The logic required to force an

initial SSL-secured HTTP request is simple. If a request does not contain a valid session ID, use `Response.Redirect` to send the browser to an `https://` URL where a cryptographically hardened session ID is generated and delivered to the client browser by way of another `Response.Redirect` pointing back to the original `http://` URL but this time with a valid session ID. Upon receiving the request, which can include a `QueryString` flag indicating that a new session ID was just assigned to the anonymous client, the ASP script might drop the new session ID as a cookie. Because the intermediate SSL-secured step can never be cached, every anonymous client is assured a unique session ID. Microsoft .NET Passport uses a similar technique for authenticated sessions. Refer to Chapter 12 for implementation details if your anonymous sessions need the ability to handle sensitive session state data and therefore require foolproof protection that doesn't inconvenience end-users. An initial SSL step combined with URL-encoded session ID solves the session ID caching problem entirely because the only unencrypted HTTP responses ever issued by the server for URL addresses that do not contain a URL-encoded session ID are redirects to the SSL-secured anonymous session creation URL. Devices on the network that cache HTTP responses will never see cache hits except when there are legitimate rerequests of the same URL with the same session ID sent by a client that is actively engaged in an anonymous session. Even if you force clients to allow cookies in order to access your site, proper management of URL-encoded session ID in addition to cookies provides you with another protection against the threat of errant caching.

JavaScript Enabled Clients Can Help to Secure Anonymous Sessions

You may be tempted to immediately drop a cookie or construct dynamic URL-encoded hyperlinks on-the-fly by creating a new session ID and sending it to the client without any special precautions. After all, if you actually receive a request from a client then you know for a fact that the client will receive the response, right? Wrong. You have no way to know whether the client is reaching IIS directly or by way of a proxy server. You can't assume that the response will be sent only to the one client you intend for it to reach, since a proxy cache could keep a copy for later use with other clients. Worse yet, you also have no way to know for sure that the client received a copy. There's nothing except proxy server programmers' goodwill to keep proxy servers from behaving badly, and if a proxy server programmer decides to send data to the client only after the entire response has been received from the server and the HTTP connection closed, there's not only nothing you can do to stop it but there is also no way to know that this happened. Either the session ID issued to the client gets used while it is still valid or it is abandoned and eventually expires. If it is used, and IIS delivered it to the client in response to an unencrypted HTTP GET request, there's no guarantee it will be used by only a single end-user Web browser.

When SSL isn't an option and you must support anonymous session state for some reason, ensuring that `Set-Cookie` headers or HTML documents with dynamic hyperlinks decorated with URL-encoded session ID are never sent to the client in response to GET requests unless the GET request includes a URL-encoded session ID provides a fair amount of protection to keep session IDs from being cached. However, there's no reason a proxy server couldn't cache responses to HTTP POST requests such that another client that sends a POST request to the same URL and with the same HTTP body could be considered a cache hit. If you can force the client to send a unique POST request

body, such as through the following JavaScript, then you can be sure that even when cache hits occur on the initial GET request that brings a new anonymous visitor to your site, the result is that clients will interpret the JavaScript and create a unique identifier to POST to the server. This way there is no chance that a proxy server will consider subsequent requests from the client to be cache hits unless the proxy server programmer writes really bad code that ignores the request body entirely and considers every POST to a URL to be the same request.

```
<SCRIPT>function postUniqueValue() { var u,d;
d = new Date();
u = Math.random();
uvForm.UV.value = d.toGMTString() + u;
uvForm.action = "default.asp?u=" + uvForm.UV.value;
uvForm.submit(); } </SCRIPT>
<HTML><BODY OnLoad="postUniqueValue();">
<FORM ID="uvForm" ACTION="" METHOD="POST">
<INPUT TYPE="HIDDEN" NAME="UV" ID="UV" VALUE="">
</FORM></BODY></HTML>
```

By returning this simple HTML document with its JavaScript-triggered automatic FORM POST as the initial response to any anonymous client request that does not yet have a session ID assigned to it, or where the session ID is determined to be inappropriate for use by the client, you create a response that any proxy or other caching device can cache without harm. Even if this document is served out of cache to every subsequent client that relies on the same proxy server, each client will interpret the script and generate a unique POST body and unique FORM ACTION. This technique must be deployed with a failover condition for compatibility with browsers that do not support JavaScript unless you intend to deny them access.

Prove-You're-Human Countermeasures

Every Web site implicitly services requests from non-human clients such as Web crawlers and automated programs unless the site's application developer explicitly detects and refuses to honor requests from such programs. Requiring authentication through any of the various authentication mechanisms described in Chapter 12 does not necessarily provide proof that a human user is directly and intentionally responsible for requests received by the server. There are any number of ways for malicious code to hijack the computer of an authentic user who has provided credentials already in order to authenticate with and receive access privileges to a Web application or restricted Web site.

The malicious code can even intercept user credentials or conduct a brute force attack to discover them. Biometric identification devices that scan a human user's fingerprint or retina, for example, are even inadequate positive proof that a human is in control of the computer making the requests at all times because in reality a human user is never in control of the computer; software is. Malicious code can easily alter the normal operation of the computer without the user's knowledge. The only way to know for sure that a human user is responsible for a particular request is to implement a prove-you're-human countermeasure that prevents the server from carrying out any processing on behalf of a

request, even one that provides authentication credentials, without confirmation from the requesting human user and a way to be sure a human supplied confirmation.

The core feature of any prove-you're-human countermeasure is that it is difficult for a computer program to respond with the correct answer or action but relatively easy for a human to do so. The easiest way to create such a countermeasure within the context of a Web application is to dynamically generate an image that contains a known numeric or alphanumeric sequence and display it to the user along with a FORM field into which the user must type the characters displayed within the image. Optical character recognition (OCR) algorithms can conceivably be used by an attacker with sophisticated attack software to respond to the challenge accurately and thereby defeat the countermeasure, but the only way to defeat this type of countermeasure is to go to such lengths. When deployed properly, the dynamic images contain characters readable to a human but not easily recognized by OCR algorithms. Various techniques such as font mixing, noise generation, and periodic changes to the visual characteristics of the images produced on the server as part of the countermeasure can defeat most OCR algorithms. You still don't have absolute proof that the client is an interactive human user rather than an automated program, but you raise the bar on automated attacks substantially. Importantly, by deploying such a countermeasure you completely defeat all JavaScript-based attacks because there is no way to analyze image content and build OCR capabilities with JavaScript.

ASP does not provide the ability to generate images dynamically without an add-on server-side object or enhanced script engine. The server side object model and scripting languages provided by default for use with ASP do not support graphics programming. However, you can read and write binary data streams using ASP with the Request.BinaryRead and Request.BinaryWrite built-in object methods. Using these methods and the built-in Application object you can prepare to generate images dynamically by chaining images together with multiple tags that your ASP script delivers to the client browser. The trick is to produce the graphic images, one per character and number in your alphanumeric character set, ahead of time and load them into Application variables using BinaryRead. The following script shows how this can be accomplished easily without using add-on objects or graphics libraries.

```
<% readBytes = Request.TotalBytes
a = Request.BinaryRead(readBytes)
d = ""
lb = LenB(a)
For n = 1 To lb
    d = d & Chr(AscB(MidB(a,n,1)))
Next
ct = Request.ServerVariables("HTTP_CONTENT_TYPE")
boundary = Mid(ct,InStr(ct,"=") + 1)
crlfcrLf = InStr(d, vbCRLF & vbCRLF)
Application("9") = MidB(a,crLfcrLf + 4,lb - (crLfcrLf + 11 + Len(boundary)))
Response.ContentType = "image/gif"
Response.BinaryWrite Application("9") %>
```

The script shown uses BinaryRead to receive an image/gif file uploaded by a browser through the use of RFC 1867 HTTP file upload. The script assumes that the GIF image being received is the one that corresponds to the number 9 in the graphic character set produced ahead of time for use with this countermeasure. You'll need to store Application("8"), Application("7"), and so on including Application("0") binary values in addition to Application("9") using additional scripts that hard-code these values like this script does or come up with a way to determine dynamically what ASCII character the graphic corresponds to that is uploaded by the browser. If your prove-you're-human graphic character set includes letters, you'll need to receive and store those binary values as well. The HTML FORM required to POST a file to the script using multipart/form-data encoding looks like the following.

```
<html><body>
<form action="imageupload.asp" method="post" ENCTYPE="multipart/form-data">
<input type="file" name="file1">
<input type="submit"></form></body></html>
```

With binary graphic data stored in Application variables for each character in your graphic character set, your scripts can now access that data and send it to Web clients on-demand. A simple pseudo random number can be used as the dynamic prove-you're-human password, and the graphic images that correspond to each digit of the password can be sent to the Web client along with a FORM that allows them to type the password that they see displayed in the sequence of graphic characters loaded by their browser. The following script does this. The hypothetical SaveU function must save the random password in the current user's session state so that it can be retrieved during the password verification step by the code that receives the FORM POST containing the user's input.

```
<% Response.Buffer = true
err = 0
done = false
c = 1
Randomize
u = Int(900000000 * Rnd + 100000000)
lenu = Len(u)
err = SaveU(u,"Password")
While err = 0 and done = false
    n = Int(9000000000000000 * Rnd + 1000000000000000)
    err = SaveU(n, Mid(u,c,1))
    Response.Write "<img src=""image.asp?u="
    Response.Write n
Response.Write "" width=50 height=50 border=0>"
    c = c + 1
    If c > lenu Then
        done = true
    End if
Wend
If err = 1 Then
    Response.Clear
```

```
Response.Write "Error Processing Request."  
Response.End  
End if  
%>
```

It's important to note that while this particular implementation of the prove-you're-human countermeasure works well enough and is very simple to build using just ASP script, it has limited usefulness for defending against sophisticated attackers. By assigning each character in the character set its own fixed image data, an attacker need only capture that data and create a table mapping the binary data for each character image to the ASCII character that it actually represents. A better way to implement the graphic character set is through the use of a dynamic image generator that produces a single graphic image that includes every character of a dynamically generated password. This way an attacker can't create a simple lookup table to determine automatically what the password is that the server is asking for. Another good way is to create a one-time-use password list and corresponding graphic images that you load as binary data into Application variables. Each time one of the predetermined passwords from the one-time-use list is selected for a particular user session, remove the binary data from the Application variable and purge its corresponding ASCII representation from the one-time-use list. This way you can still use the technique shown here for full prove-you're-human security rather than deploying a dynamic image generator feature to your server. Figure 7-2 shows a prove-you're-human challenge sent to the client when the password is 612483957.

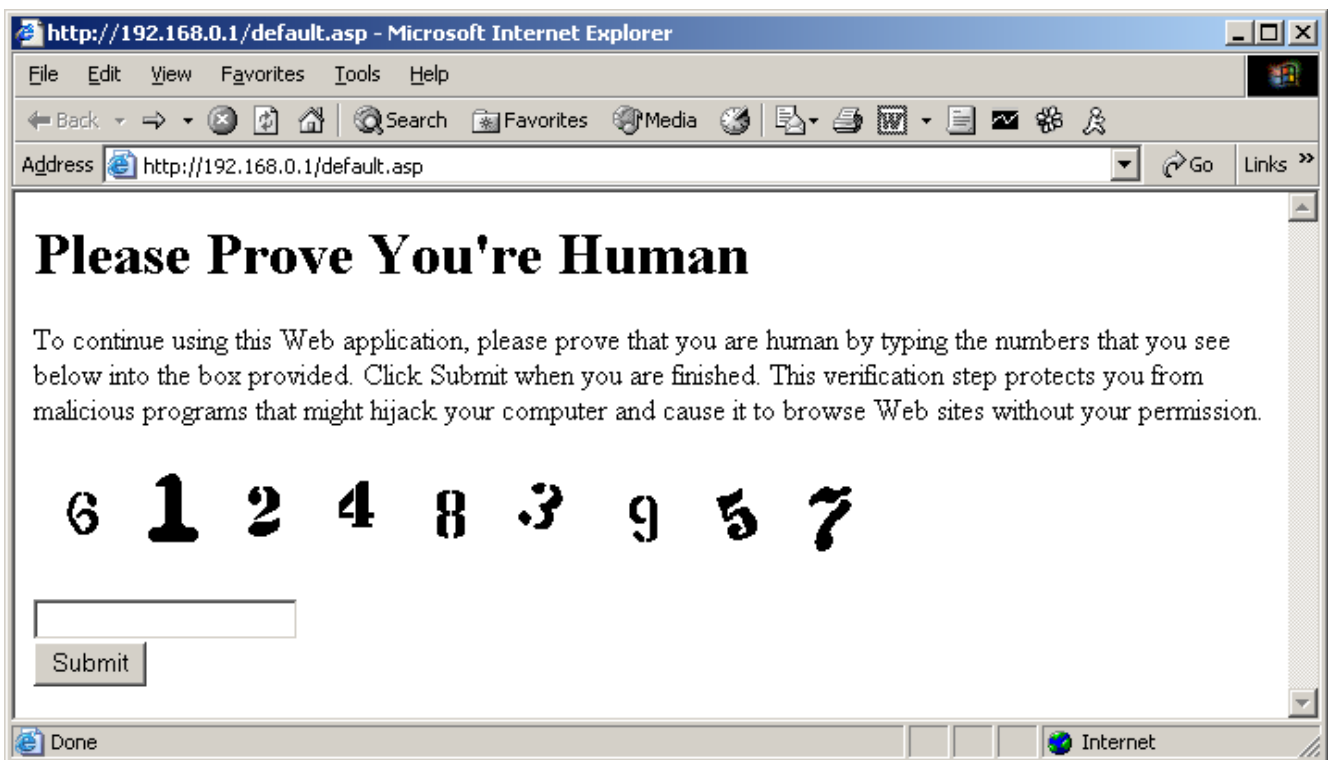


Figure 7-2: Prove You're Human or Go Away

After the dynamic `` tags are sent to the client, it sends an HTTP request to the server for each image. The URL references a script rather than an image, so the script

must determine which character to send as binary image data based on its knowledge of the password generated in the previous step and the unique number encoded in the URL that prevents caching and hides from unsophisticated attackers the character that must be supplied as part of the password. The following script is placed in image.asp to take these steps, setting ContentType to image/gif and using BinaryWrite to send graphic image data for each character to the client. A hypothetical function called LookupU is responsible for translating the unique number assigned in the previous script to the correct ASCII character in the password, which also happens to be the key used to name the binary data Application variables stored previously. The LookupU function reads session state values created by the previous step, so the details of this function are dependent on how the application manages session state storage as are details of the previous SaveU function.

```
<% Response.ContentType = "image/gif"
n = Request.QueryString("u")
If IsNumeric(n) Then
    c = LookupU(n)
    Response.BinaryWrite Application(c)
End If %>
```

By placing this script in image.asp, and referencing image.asp with a QueryString parameter as the src of an image, you enable ASP to dynamically select the right binary graphic data and deliver it to the client out of an Application variable. The script that receives the FORM POST (not shown) must retrieve the random password that was stored in session state previously by a call to SaveU(n,"Password") which is an arbitrary variation of the hypothetical SaveU function call. You could write a different function for saving the password to session state, or overload it as the scripts shown here presume.

LookupU could likewise be overloaded to retrieve the password when passed the word "Password" as a parameter rather than a numeric string containing a random number assigned previously to one of the password characters. It is this lookup step that enables the HTML to reference multiple images, one for each character, without revealing anything about the content of each image.

The simplistic ASP implementation of a prove-you're-human countermeasure shown here doesn't require an attacker to use OCR to read and decode the image data, since the implementation shown uses the same fixed graphic data for each character and delivers each character as a separate image to the client. To raise the bar for a successful attack to force it to use OCR requires a different approach to creating and delivering the graphic image data to the client. However, if you're satisfied to simply defeat malicious JavaScript attacks that might hijack Web browsers of your Web site users and force those browsers to send requests to your server, the approach shown here works quite well. It is particularly resistant to script-based automated clients if you carefully construct each graphic image in your character set so that it has the same length, in bytes, and the same width and height as every other character. Visual Basic Script version 2.0 introduced a script function called LoadPicture that has the ability to read binary data for images into an object that implements the IPicture COM interface. Although it doesn't give a script access to the image data, LoadPicture can be used to determine width, height, and size in bytes. Designing your character set so that these characteristics of each image exactly

match every other image makes the character set fully resistant to script-based attacks unless the scripts are able to invoke and control code that has additional image processing abilities.

Development tools often have a simple way to automate RFC 1867 HTTP file upload, so the process of populating Application variables with graphic binary data for each character in your character set can be automated for simplicity. It goes without saying that permission to upload (and thereby define) the graphic characters used in your graphic character set should be granted to administrative staff only. Differences in the length in bytes of graphic data or the width and height of the graphic image are the easiest ways for an attacker to automatically circumvent this type of countermeasure, so giving the attacker the ability to define binary graphic data for characters in the character set is a bad idea. Unless your Web application is an XML Web service meant to be used by automated clients who are programs not people, you should restrict access to your application to humans-only through the use of a prove-you're-human countermeasure like the ones described in this section.

Garbage In Garbage Out

Valid FORM POST requests sent in connection with legitimate anonymous sessions carry with them a special risk. There's no way to know that data provided by the anonymous request is legitimate, and nobody to hold accountable for sending garbage to your server when it's not. The countermeasures required to defend against this threat depend on your application design and can't be adequately explained in detail using general terms that would be appropriate for this book. Making database updates possible for anonymous sessions means you are deciding to let anything that looks valid into your database. Don't be surprised when you end up with junk data.

One entertaining example of inappropriate trust of the legitimacy of FORM data sent to a Web application by anonymous users comes from the requirement under U.S. law for people who send unsolicited e-mail messages using electronic mailing lists to provide recipients with a way to opt-out of the sender's list so that they won't receive future mailings. These opt-out mechanisms are often built around Web servers that allow anonymous sessions. When you receive spam and you want to opt-out of future mailings from the sender, you are often asked to visit a Web site where you can provide your e-mail address in order for it to be removed from the list used by the sender for future mailings. Luckily, spammers are dumb (and ugly) so they don't bother to assign each e-mail recipient an anonymous session ID inside the spam then require that this session ID be sent back to them in a reply or by way of a Web site in order to accept an opt-out request. By allowing any e-mail address to be provided to their opt-out Web page, they give you the ability to opt-out all of your friends and family and everyone else on your copy of the latest "600 million fresh e-mail addresses on CD-ROM" that you received as a birthday gift from a relative who thought it would help you market your company. Anti-spam activist groups are known to use these lists on CD-ROM to send opt-out requests to anyone who will accept them through anonymous sessions. Of course none of this stops the flood of spam sent maliciously in violation of law without legitimate opt-out mechanisms that the spammer will abide by honestly. Spammers are evil and dishonest people.

Principle of Least Astonishment

In science, the principle of least astonishment states that the least astonishing explanation is usually the right one. This principle is similar to Occam's Razor which states that simpler theories are preferred over more complex ones. The principle of least astonishment has been adapted by engineers to the issue of product design with the meaning that the right way for a product to function, even when it does something wrong or deals with an error condition, is the way that least astonishes the user. When a user enters a password that exceeds the maximum length allowed for passwords, for example, the principle of least astonishment would hold that the software should not accept the user's password silently without telling them the password was too long for the software to use because the user would later be astonished when they are unable to login properly. Users should always be able to understand unexpected results even when errors occur. Receiving access to another user's session state, even when that session state belongs to another anonymous user, would violate the principle of least astonishment. When in doubt whether or not a feature is secure, consider whether it will ever astonish the user (or the developer). Remember Murphy's Law: that which can go wrong most likely will go wrong.

Cross Site Scripting

Cross Site Scripting, known as XSS rather than CSS because the abbreviation CSS was already used by Cascading Style Sheets and XSS is such a complex and pervasive security threat that confusion with CSS caused the infosec community to assign this class of threats a new more distinct moniker. In short, XSS is any flaw in the trust model used by software that causes it to inappropriately trust a particular network node, a remote service, or remote code. XSS is typically accomplished through the delivery of script instructions over the network that allow a malicious third party to exploit the misplaced trust, which results in some distinct vulnerability such as data theft, a privacy violation, or malicious code injection. Generally, XSS is a vulnerability that is almost completely eliminated through the simple act of disabling client-side scripting and other programmable content such as ActiveX Controls that a remote server can cause to execute inside a Web browser process running locally on the client computer. If everyone in the world used Web browsers that only allowed static HTML content without code, almost all XSS threats would be eliminated instantly. Obviously this isn't going to happen, so you must take steps to protect users from XSS exploits.

A common example of XSS results in the content of browser cookies being revealed to sites other than the ones they are supposed to be associated with exclusively. Another result could be the injection of malicious client-side JavaScript into a Web browser that is in possession of a cached authentication credential or session identifier and is therefore granted elevated privileges for access to particular Web site content or services. The injected JavaScript can force the Web browser to retrieve information or send commands to a server that an attacker's Web browser wouldn't be capable of without the cached credentials, giving the attacker increased client privileges. With simple JavaScript exploit techniques like opening new browser windows behind the active window (a so-called "pop-under") or creating hidden frames, malicious script injected through XSS can force the browser to be a proxy server of sorts on behalf of the attacker. Even when firewalls are used to prevent an attacker from sending unsolicited packets to a victim's computer, nothing prevents the malicious JavaScript from polling the attacker's computer

periodically through outbound HTTP requests that the firewall doesn't block in order to find out what the attacker wants the hijacked browser to do next. Outbound HTTP requests can also be used by malicious JavaScript to deliver stolen information or malicious scanning results to the attacker.

One particularly interesting XSS technique that attackers commonly use that demonstrates the widespread real-world threat of XSS and the importance (and difficulty) of defending against it works as follows. The attacker knows that certain people trust your Web site based on its DNS domain. When users see your DNS domain in a URL, they automatically assume that it's safe to click on that URL because your domain couldn't possibly host malicious content, right? Put aside the reality that this is a flawed assumption for users to make for the sake of understanding the XSS threat. Yes, a malicious third party could hijack your Web server, hijack your DNS, act as a Man In The Middle with respect to the victim user, or do other things to send malicious content to the victim's computer instead of authentic content produced by your servers. When an attack uses any hijacking or spoofing technique, it isn't XSS it's something else. Now back to the user and the trusted URL.

When the user's Web browser sends a request to your authentic Web server, and your Web server has not been hijacked or compromised in any way, you may think there's no way for an attacker to force your authentic Web server to deliver malicious content to the user's Web browser. But you're probably wrong. Few Web servers have been completely hardened against XSS attacks, partly because so many people misunderstand the risk posed by XSS and partly because every request processed by a Web server can potentially result in an XSS vulnerability. And every line of server-side code written by a Web application developer can potentially contribute to XSS. The Web browser receives malicious content of the attacker's choosing delivered by your authentic Web server because the URL encodes the malicious content, or a FORM served up on one domain is assigned a POST ACTION that points to your Web server, and your server echoes the malicious content back to the client browser unchanged or decoded. When the malicious content moves from URL-encoding or FORM POST-encoding inside the HTTP request body to the body of a document loaded by the browser, the XSS attack is successfully launched and the damage is done. Many Web servers are vulnerable to XSS in the default configuration because they send unfiltered information about the bad request made by the client browser in the content body of Web pages that display error messages. In this case every invalid URL and every URL an attacker can discover that points at your server and results in an error message can be used to fool users who trust your DNS domain into using URLs that result in XSS attacks. Figure 7-3 shows how to disable detailed error messages in IIS using the MMC interface.

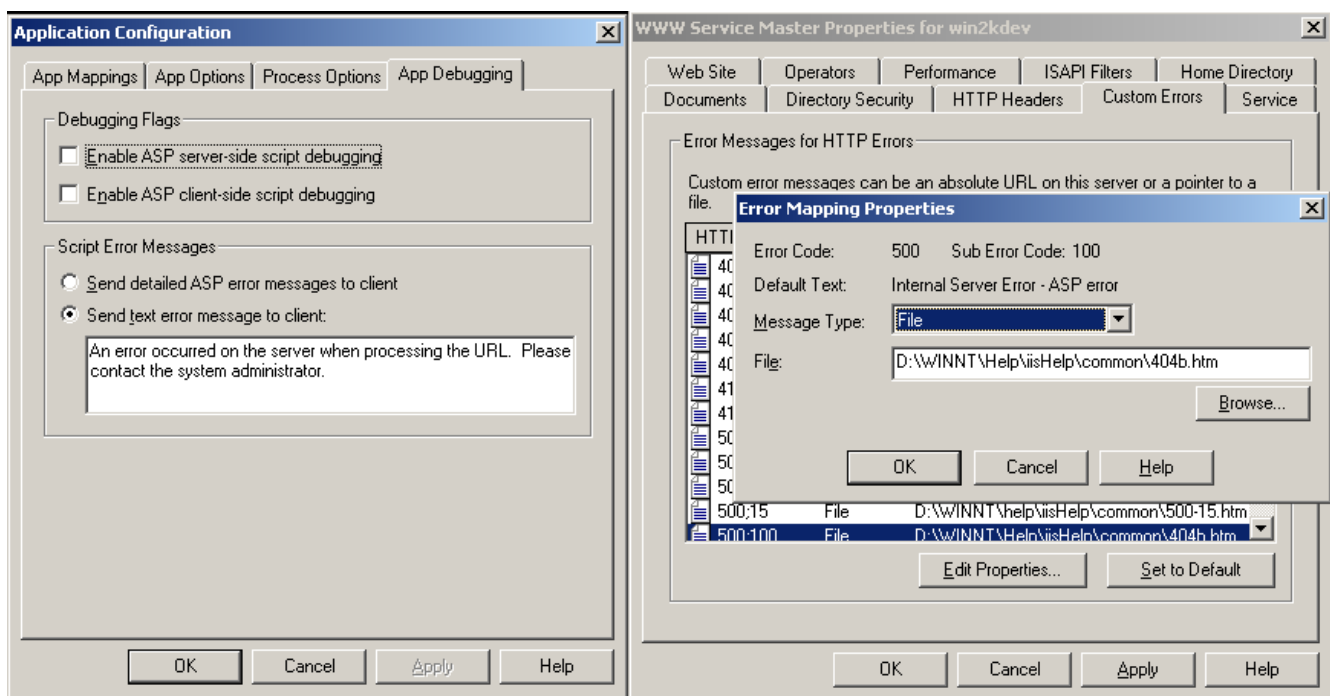


Figure 7-3: Disable Detailed Error Messages in ASP Applications

Disable detailed ASP error messages in the WWW Service Master Properties so that every new WAM application instance created under IIS on the box will share the same default setting. Locate the Application Configuration window by selecting the Home Directory tab and clicking the Application Settings Configuration button. In addition, you must edit Custom Errors as shown in Figure 7-3 to change the setting for error codes 403;14 “Forbidden – Directory Listing Denied”, 500 “Internal Server Error” and 500;100 “ASP Error”, as well as error codes 501 “Not Implemented” and 502 “Bad Gateway” so that they do not use IIS Default error handling which reveals far too much information about the server configuration and script source lines that may be useful to XSS exploits or human attackers. By setting each of the Custom Errors to a static HTML file like 404b.htm as shown, you disable Default error handling and prevent XSS as well as sensitive data leakage such as script source.

XSS in Services Running Alongside IIS

The more people who use and trust your Web site and your DNS domain the more damage an XSS exploit can do and the more victims it can reach. Flaws in your network services other than IIS can also result in XSS. The Echo port (TCP/7) is the easiest of all to exploit for XSS because the echo service is designed to echo data it receives back to the client browser, which is the essence of any XSS attack. If you have Simple TCP/IP Services installed and active on your Windows box, you can try the following XSS exploit URL. It includes a total of 1,033 characters. The minimum number of bytes necessary for Internet Explorer to immediately interpret HTTP response body data as it arrives rather than wait for the TCP connection to close as an indication to the browser that it has received all of the data the server is going to send is 1,025 because Internet Explorer uses a 1,024 byte (1 kilobyte) receive buffer that must fill up before action is taken on the client side to interpret and attempt to display or use network content.

Now, the 127.0.0.1 loopback IP address example URL shown doesn't give the malicious script any real XSS abilities because you probably don't provide services to yourself on the loopback address. However, replace the IP address with a valid FQDN that resolves to a box that does provide Web application services in addition to the TCP Echo service and you've got XSS. Hopefully this is enough of a demonstration to show that the Echo service should be disabled on any box that runs IIS. Remember also that if IIS is behind a port forwarding NAT router or proxy server that itself exposes any services such as the Echo service, disabling this service on your IIS box accomplishes nothing because clients elsewhere on the network will still receive Echo services from the IP address used to contact your IIS box. This is all that matters from the client's perspective in order to facilitate XSS; there's no requirement for the box that echoes data back to the client to also be the box that provides application services. Any time the client perceives different physical entities on the network to be the same network node, or to be part of the same DNS domain, the potential exists for XSS exploits to occur.

XSS in Active Server Pages Script

In the last chapter you saw how ASP.NET can automatically reject requests for files that aren't supposed to exist or that have been modified without permission on your server by using a hash code validation layer. This offers a good defense against Web application tampering, but it does little to address XSS threats that exist in nearly every unhardened Web application. Any time a server-side script receives data and sends it back to the client unfiltered, a Web application's users are vulnerable to XSS. Both HTTP GET and POST requests are potentially vulnerable when the server scripts the dynamic handling of variable data or when the URL is intentionally malformed. Take the following ASP script as an example of what not to do.

```
<% Response.Write "<p>Thank you for registering. Your user ID is " &  
Request.Form("userid") & ". Your password has been sent to you in e-mail. Have a nice  
day.</p>" %>
```

Because Request.Form("userid") contains unfiltered data submitted in the HTTP POST request, Cross Site Scripting is possible against users of the application. An attacker could send an HTML-formatted e-mail message containing a FORM with an ACTION set equal to the URL of a script that contains an unsafe Response.Write like the one shown and by placing a hidden FORM field named "userid" in the e-mail message that contains malicious JavaScript, any e-mail client that allows HTML-formatted e-mail can serve as an attack vector for this type of XSS. When the e-mail client allows JavaScript also, there is even a potential for automatic script launch. The JavaScript could automate a POST request, forcing it to be sent to launch the exploit.

Automatic Launch Vectors for XSS

Cross Site Scripting doesn't require a user to make a foolish mistake. It simply requires an attacker who is capable of influencing where a user's computer sends HTTP requests. Safe computing practices are not enough to protect against XSS, the burden of protection can't be placed entirely on end-users. By exploiting known vulnerabilities in the way that certain Web browsers identify MIME types of multi-part documents, an attacker can turn what looks like a URL for a harmless graphic image into an XSS exploit. And users don't

have to click URLs in order for browsers to send requests to them. A properly executed exploit can use an HTML tag to launch XSS in certain browser versions, and large numbers of users are vulnerable whenever Web site defacements occur. An attacker who gains the ability to edit Web site content can launch all sorts of attacks including XSS exploits against victims who browse to the affected site expecting to find benign content. HTML formatted e-mail messages can also launch XSS, with particular ease if client-side scripting is allowed for HTML formatted e-mail messages received by the e-mail client program. Any time code, including script, is allowed to travel over the network, it becomes an avenue of potential attack automation. Content types such as Microsoft Office documents that allow embedded code also facilitate automated XSS launches.

Input Validation Rules

The difference between securing IIS and securing a Web application hosted by IIS centers around ensuring that any feature enabled by the application is fully hardened against attacks and is free of security-related bugs. By default IIS will not process input sent in FORM fields or QueryString URL-encoded name/value pairs, it simply ignores such additional input and attempts to locate the resource identified by the URI base path. When unnecessary name/value pairs are supplied in a request, they have no effect unless the request invokes a script engine, ISAPI filter, ISAPI extension, or executable CGI-compliant program and that request handler does something with the additional information. Buffer overflow vulnerabilities in any request handler can potentially cause IIS to allow something to happen due to the presence of unnecessary malicious data in a request, but this isn't the same as IIS or a hosted application purposefully using the data. Interpreting and using variable data supplied in HTTP requests is normally the job of a Web application. Everything that an application does to process variable data when handling requests that contain it must be carefully scrutinized for security risks. Input validation means storing variable data in variables and applying filters to sanitize all variable data before it is used. The badly-written Response.Write ASP script line shown previously can be rewritten as follows.

```
<% var = Sanitize(Request.Form("userid"))
If IsValidUserID(var) Then
Response.Write "<p>Thank you for registering. Your user ID is " & var & ". Your password
has been sent to you in e-mail. Have a nice day.</p>"
End If %>
```

The hypothetical Sanitize function referenced here would apply whatever generic filtering rules are appropriate for your application. Typically that means removing anything that is not alphanumeric, but the requirements of your application may differ. Before using information for a specific purpose, such as displaying a user ID as shown in the example, another validation function (e.g. IsValidUserID) should be called that is specific to data presumed to be present in a variable. Both functions are necessary for safe scripting. Sanitize and validate, preferably in that order.

Assume Purposeful Web Client Tampering

Malicious input is possible in cookies, URLs, HTTP headers, the HTTP request body, and nearly any other aspect of client interaction with IIS through the use of a conventional Web browser and a text editor. Custom programs to help with network intrusion or hacking aren't necessary to damage or exploit Web applications that make invalid assumptions about the trustworthiness of data received from HTTP clients. The simple defense tactic is to assume that any data originating from the network is hostile until that data has been validated, sanitized, and forced into the one and only character encoding that is considered valid for representation of the bytes.

Persistent cookie files stored on the end user's hard drive can easily be edited by hand with any text editor. The Web browser will transmit whatever it finds in the cookie file as a regular Cookie: HTTP header. Every HTTP response body can be captured in raw form through the view source option in the browser or a network sniffer. When SSL is used, debuggers and other tools that are able to dump memory contents of a process as well as custom-coded SSL-compatible clients also have access to HTTP response body contents, which gives full access to any hidden information embedded in scripts or FORM fields. Access to the HTTP response body also reveals every FORM field name and id, allowing a malicious user to save and edit the body with a text editor so that it can be opened locally. In this way any user with a text editor and a little knowledge can interrupt normal Web browsing at any time and tamper with the values sent by the browser to your Web application.

Respecting the privacy of users must be absolute. Rampant dropping of persistent cookies causes users' computers to accumulate a trail of cookies that another person can examine to see where the user has been on the Internet. It is unreasonable to expect users to understand and actively manage the storage and privacy of persistent cookies therefore users must be empowered to choose if, when, and for how long persistent cookies get created.

All an end user has to do is view source in their browser or save a Web page to a file and they can see, and edit, every aspect of the HTML and client-side script contained in the page. For this reason it is never acceptable to place information such as the price of a shopping cart/product catalog item or the amount a user's credit card will be charged when they complete an online purchase into a hidden FORM field or a client-side script variable. You can never trust the client to tell you the truth, so it follows that relying on the client to tell you the truth is never an acceptable design feature of a Web application unless there is no need for security and data integrity.

Harden Session Identifiers

Session identifiers can be forged, whether they are stored in URL-encoded name/value pairs, cookies, hidden form fields, dynamic URL subdirectory paths, or some place less common that still enables the client to receive, store, and transmit with each request the session ID assigned to it. SSL encryption does not prevent forged session ID values from being sent to the server by an attacker, because an SSL-secured server will always accept encrypted requests and attempt to respond to them. The HTTP request may be encrypted, but that doesn't mean it isn't malicious. Although by definition unique values, session identifiers must be more than just unique in order to be secure. They must be resistant to brute force attacks where random, sequential, or algorithm-based forged

identifiers are submitted to the application until a valid identifier is found. They must appear random so that guessing a valid session ID is no easier than brute force hacking. And, ideally, valid identifiers will bear some sort of digital signature in addition to a random component.

By hashing the session ID and then encrypting the hash with a secret key and attaching the encrypted hash to the random component, you make a session token that is both a random session ID and a digital signature. The digital signature proves the creator of the session ID was in possession of the secret key at the time the token was created. You can also place additional fields of information in the encrypted portion of the session token such as the IP address of the client to which the token was originally issued. Either treat a change of IP address as fatal to the integrity of the session, or allow a range of addresses to use the signed token. The address range technique accommodates the real-world existence of client side proxy farms and dialup DHCP address pools where a client's IP address may appear to change periodically, or even with each request. An attacker who forges a valid session token may in this case be unable to make use of it because the source IP address of the attack does not fall within the authorized IP address range. Unless the attacker is in control of a computer located inside the authorized address range, of course.

Session identifiers that are truly random (created with a hardware random number generator rather than a pseudo-random number generator) and also long enough to withstand brute force attacks may be just as difficult, or even more difficult, for an attacker to guess. But relying on session ID values to be random places too much trust in the random number generator. A flaw in its operation or an attack algorithm that is able to predict the values it produces with more accuracy than pure chance can break security that is based on such systems because they represent a single point of failure vulnerability.

There is no harm in also digitally signing the random value, and doing so increases the length of the resulting session token. It adds a second point of protection that also must fail in order for security to be compromised. It is far less likely that an attacker will both predict a value produced by a random number generator and deduce the secret key used to digitally sign that random number. Even if the random number generator is flawed or intentionally compromised by malicious code, session tokens that include digital signatures are still secure and resistant to brute force guessing.

It is important for both privacy reasons and cache prevention reasons to prevent IIS/ASP from automatically dropping session cookies. A registry switch is used to disable IIS/ASP Sessions. See Knowledge Base Article Q163010

Another benefit provided by digitally signed session tokens is that custom client programs can verify the digital signature explicitly if they are in possession of the public key that corresponds to the private key used by the server to apply its digital signature. SSL is not conducive in its real-world design and deployment to the job of positively identifying the server as a particular known entity. Certificate trust chains are prone to tampering or flaws, and are usually too open-ended so that any number of root authorities could have vouched for the authenticity of a server identity other than the one root authority that you expect. It is safer to rely on digital signatures applied to session tokens for explicit positive identification of a server than to rely on SSL to do this for you automatically, and

doing so is not difficult. Conventional Web browsers don't have the ability to verify digital signatures applied to application data stored in session tokens, they are limited to server authentication provided by SSL using the root certificates configured on the client system. However, custom clients created with a programming tool that simplifies cryptographic programming like Microsoft .NET can easily verify digital signatures contained in application data. This is especially important as a technique for securing XML Web services by enabling the clients that depend on them to more adequately authenticate the identity of the remote server that is providing the XML Web service.

Impose Explicit Character Set Encoding

Most Web applications know ahead of time what character set is valid for data input. Allowing any other character set is not only unnecessary it's also dangerous. However, explicitly requiring data that is sent to the server to conform to a character set like ASCII is not enough, you must also filter data to an authorized subset of characters that exist in the character set. Malicious JavaScript is written entirely in ASCII, for example, and to filter data input so that it can't contain JavaScript is important if JavaScript is not a valid component of authorized variable data input. Whenever possible, limit data input to specific characters rather than attempting to scan for character sequences like "<script>" as a way to filter out bad input. The following ASP script sanitizes FORM field input. The script throws out characters that are not letters or numbers, so that "<script>" will be converted to "script".

```
<% Function Sanitize(input)
Dim o, a, b, c, lenin
lenin = Len(input)
For a = 1 to lenin
b = Mid(input,a,1)
c = Asc(b)
If (c >= 65 and c <= 90) or (c >= 97 and c <= 122) or (c >= 48 and c <= 57) Then
o = o & b
End if
Next
Sanitize = o
End Function %>
```

In addition to sanitizing input that your ASP scripts send back out to clients immediately, you should always be careful not to store malicious content in your database. If you don't sanitize content before it enters your database, it can have a malicious impact at a later time when it is retrieved and used. In addition, every SQL statement that you construct dynamically based in part on user input or other variable data carries a special risk. Arbitrary SQL statements can be injected into such dynamic SQL unless your script carefully removes any SQL syntax or other malicious variable data that could be interpreted by your database server.

Safe Scripting for Internet Applications

Scripts that are accessible from Internet-based clients have more to contend with than those that are part of intranet-based applications. The reality of proxy farms and aggressive

cache policy, the uncontrollable and unknowable nature of global Internet routing, the inclination of real-world Web sites to link to each other and share resources, business information, and visitors complicates security immeasurably. In addition to being more explicit about cache prevention when it would cause your Web application harm, and being more sensitive to privacy and the information leakage risks posed by various Internet business practices, it's a very good idea to impose strict verification procedures to codify any assumptions that you make. For example, an application that is deployed to an IIS box that has an SSL certificate installed should force SSL to be used by clients of ASP scripts that expect SSL rather than just assuming that SSL will be used. The final three sections of this Chapter delve into these issues, which are especially important for Internet-based applications.

HTTP and HTML Cache-Busting Techniques

Cache is a constant threat to Web applications for several reasons. The most severe threat is the potential for dynamically-generated Web pages containing private information meant only for a particular user to fall into the hands of others because of storage in cache. HTML META tags are one way for content caching to be influenced by content authors. For example, the following META tag specifies the expiration date and time of an HTML document. Programs such as Web browsers won't disallow access to the content after its expiration date, but well-behaved proxy servers and other network devices that manage cache are supposed to comply with expiration dates whenever and however they are made available by content owners and authors.

```
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">  
<META HTTP-EQUIV="Expires" CONTENT="-1">
```

Carefully guiding client browsers (and proxies that often facilitate client access to servers) as to how long each item delivered by the server should be cached is very important. The HTTP header "expires" and the HTTP 1.1 header "CacheControl" should both be set at all times in pages that should never be cached by either proxy servers or the client browser. The following ASP script lines should then appear all over the place, they can never be used too often.

```
<% Response.CacheControl = "no-cache" %>  
<% Response.AddHeader "Pragma", "no-cache" %>  
<% Response.Expires = -1 %>
```

If no-cache is specified in Response.CacheControl, Internet Explorer won't save the contents of the HTTP response to the Temporary Internet Files folder. When the back and forward buttons need to function for usability of a Web site, you should not specify no-cache but instead provide only an Expires header and META tag. Proxy servers that are well-behaved will comply with the immediate expiration requested, but client browsers will still behave as users expect them to with respect to the back and forward buttons and automatic local temporary caching.

For more on cache control see Knowledge Base Article Q238647 at <http://support.microsoft.com/support/kb/articles/Q238/6/47.ASP>

For information on legal protections afforded to service providers and Web site operators in the U.S. by the Digital Millennium Copyright Act specifically to address the legal threat of uncontrollable copyright violations when cache or other automated systems result in unauthorized copying and distribution, see Library of Congress DMCA directory of ISP agents <http://www.loc.gov/copyright/onlinesp/>

Setting expires and cachecontrol HTTP headers to ensure immediate expiration and private cache management of the response that initially sets or re-sets the sessionid cookie helps to reduce the likelihood that more than one browser will be assigned the same session. Assuming that multiple browsers may in fact have been assigned the same cookie value is also an important security policy. Whenever possible, abandon the assumption that an unencrypted HTTP session that has relied on an anonymous session identified by a session ID and explicitly authenticate and issue proper authenticated session tokens to users who move from anonymous browsing to sensitive activities that must not fall victim to either intentional or unintentional session hijacking.

URL-Encoded Name/Value Pair Disclosure

The HTTP header named Referer: (sic) is provided by Web browsers in HTTP requests sent to sites linked from the current page. The full contents of the current URL including name/value pairs becomes the Referer – the address from whence the Web client came. This presents a serious complication for securing a Web site that allows URL-encoded data of any kind. Every request sent by the client browser to URLs linked from the current page can potentially leak sensitive information such as session ID or the content of FORMs that use the GET method rather than POST. In ASP script the built-in Request object gives access to the Referer HTTP header.

```
Request.ServerVariables("HTTP_REFERER")
```

The Referer HTTP header combined with URL-encoded name/value pairs will even result in sensitive information leakage to Web sites of advertisers who host their own advertisement graphics that are referenced within a page using an tag. It isn't possible (or desirable) in most applications to completely eliminate URL-encoded name/value pairs. For one thing doing so prevents session state tracking for users who disallow cookies. With careful management of the risks of leaking encoded name/value pairs such as through the creation of a "launching pad" script to which any hyperlink to a foreign site is implemented, privacy and security risks of URL-encoding can be mitigated. Remember, however, that extra care must be taken for any area of a Web application that allows users to send HTML content to the server. All it takes is an tag embedded in that HTML that references a foreign URL and a brand new information leak can be created.

Forcing SSL Access To Scripts

ASP scripts that are designed to process requests for SSL-secured Web sites may end up being activated through HTTP requests that do not use SSL. One reason an attacker might want to make this happen intentionally is to mount an XSS attack using a FRAMESET that references URLs from both an SSL-capable HTTP server and one that is not SSL-capable. If the attacker mixes http:// and https:// URLs in the same Web page

a warning message will be displayed to the user about mixed secure and unsecured content. To avoid this warning message, it may be desirable for the attacker to change https:// URLs to http:// instead. If your ASP script is designed to process requests that are encrypted using SSL, it's a good idea to make SSL an explicit requirement for access to the script. The following shows how this is done.

```
<% If Request.ServerVariables("HTTPS") = "off" or  
Request.ServerVariables("SERVER_PORT_SECURE") = "0" Then  
Response.Clear  
Response.End  
End if %>
```

ASP scripts can also impose minimum key length requirements for both the symmetric secret key used for bulk encryption (the block cipher, typically the RC2 algorithm) in an SSL connection and the length of the asymmetric private key used by the server. ServerVariables HTTPS_KEYSIZE and HTTPS_SECRETKEYSIZE provide these length values, in bits. The server's asymmetric private key corresponds to the public key that is digitally signed and contained in the server's certificate issued by a CA. The value of HTTPS_SECRETKEYSIZE depends on the number of bits selected for the RSA key pair generated for use in SSL by the server and may be one of 512, 1024, 2048 or another valid key length for the RSA algorithm. The value of HTTPS_KEYSIZE will be 128 when 128-bit encryption is in use, and 56 when 56-bit encryption is the highest encryption level available in the browser. Old export restrictions on encryption software in the U.S. capped early Web browsers released by U.S. companies to other countries at 56-bit encryption while Web browsers shipped to U.S. customers were given the ability to use higher encryption.

Writing secure Web applications isn't difficult if you're aware of the most common mistakes and historical vulnerabilities and design secure applications consciously. There is no reason for any Web site to expose Cross Site Scripting flaws that leave users' exclusive control of their private information and active sessions open to attacks with the simplest of client-side JavaScript. When Internet users and developers decided to allow, as a matter of standard practice, pieces of script code and even compiled machine code to travel over the network and be used automatically by client programs like Web browsers, additional security burdens were automatically created for everyone. These burdens are possible to manage properly with knowledge and rigorous monitoring of vulnerable clients and servers. However, these are burdens created for all users by decisions made by a relatively small number of programmers whose primary concern at the time was not information security. This shows us that any time code is written and deployed on the network it can have widespread security implications that transform harmless artifacts of everyday programming, like code that gets put into production without adequate testing and security analysis, into security problems for everyone who uses the network.

The input validation and security hardening techniques shown in this Chapter touched on the most important aspects of designing and developing secure Web applications. Properly managing anonymous sessions and designing Web application code with a full appreciation of the bad things that the network can and will do to your application and its users will enable you to avoid most of the problems normally associated with Web application security. Remember that every line of code written under any hosted

application can create new vulnerabilities for IIS, its other hosted applications and Web sites, and anyone who uses the network. The one assumption you can always make with safety when developing or managing Web applications is that there will be security flaws unless somebody does something to prevent them.

Chapter 8: TCP/IP Network Vulnerabilities

Packet routing in a public data network is inherently untrustworthy. You can be reasonably sure that your own hardware and network wiring (or wireless encryption) are trustworthy, provided that you manage physical access to the equipment and secure it from remote access with a firewall and password protection. But you have no control over the security measures implemented by your Internet Service Provider and other people's networks therefore every packet that originates from the Internet must be considered malicious until it can be authenticated. TCP/IP traffic commonly flows in both directions based on presumptions of trust that are invalid. You may not be able to eliminate all such untrustworthy traffic, but you must know where it exists in your network, the networks of users, and Internet Service Providers, so that you can carefully weigh its impact on security for your IIS box.

Tracing The Flow of Trust in TCP/IP Networks

Routers, proxy servers, firewalls, and even hubs are vulnerable to attacks that compromise the trustworthiness of any TCP/IP network, not just one that routes traffic to and from the Internet. Any device that performs a function on the network is dependent on other devices and this dependency creates a web of trust that must be protected through administrative security policy and by programmers who know the vulnerabilities and code around them purposefully.

As packets flow through the network, so too does implicit and explicit trust. For example, consider that TCP/IP clients usually depend upon a response from a Domain Name System (DNS) server to decide the IP address to which to send packets in order to contact a particular server. The client has no way to know whether the DNS lookup resulted in a legitimate IP address, it simply assumes the response to be legitimate. Likewise, TCP/IP routers, hubs, and switches typically offer no security and presume that the devices connected to them are authorized and authentic.

Domain Name System Vulnerabilities

The original Domain Name System (DNS) doesn't include any security mechanisms and this spells a recipe for disaster because TCP/IP clients commonly offer authentication credentials to remote servers based on the assumption that the DNS server tells the truth and the IP address received from the DNS server is the legitimate remote server address. Tampering done by a malicious third party that impacts DNS lookups can result in user IDs and passwords being sent by clients to servers controlled by the third party. Clients should not explicitly trust servers simply because they are forced to implicitly trust the validity of DNS lookup results received from DNS servers, but they nonetheless do.

DNS is defined by RFC 1034 <http://www.ietf.org/rfc/rfc1034.txt>

DNS was not designed to be reliable and trustworthy: it contains no encryption, authentication or repudiation facilities, so any use of DNS as a means of establishing trust is faulty. The most disturbing misuse of DNS as a trust mechanism is the common

policy of Web site operators to e-mail passwords to users who have allegedly lost theirs or to ask a user to prove their identity by responding to an e-mail message. SMTP servers determine where to route e-mail for a domain based on a DNS zone table entry known as a Mail Exchanger (MX) record. Any third-party who succeeds in editing the zone table for a DNS domain is in complete control of the domain's e-mail delivery. Network Solutions (now a Verisign company) was notorious for deploying faulty trust in a critically-important component of the public commercial Internet, having built a system of authentication to control domain registrations and transfers around e-mail messages rather than a real authentication mechanism such as a user ID and password. Even today, any malicious third-party who hijacks a DNS server that provides DNS service for domains registered through Network Solutions' legacy registrar has complete control over the domains and can easily transfer the domains to another party without authentic permission.

Another domain registrar, register.com, also deployed a Web site feature that used e-mail as a trust mechanism, allowing any user of their Web site to request that an allegedly lost password be sent to them in e-mail. This vulnerability was worse than you might think, however, since abusing it didn't require hijacking of any DNS server, rather it facilitated DNS hijacking at register.com of any domain registered through register.com by allowing an unauthorized third party to login using the credentials of an authentic registrant for the purpose of making changes to DNS registrations. If an attacker already has the domain hijacked, the MX record can be modified so that register.com's lost password e-mail will end up in the hands of the hijacker – but that wasn't necessary with the register.com vulnerability because the e-mail message sent by the server to remind the user of a lost password had a URL in it that the user could click on to login automatically and select a new password. The URL contained a unique number that would tell the server which user account to grant access to when a Web browser contacted the server and requested the URL. An attacker who could guess one of these unique numbers could login automatically using somebody else's real user account and immediately change the user's password.

Worse yet, an attacker could request that register.com send them a password reminder e-mail message in order to find out what the most recent unique number was that the server had assigned to lost password requests and then search backwards from that unique number to find other valid unique numbers. To hijack a particular domain, an attacker needed only to fill out the lost password form at register.com on behalf of the domain registrant so that a unique number would be assigned to the automatic login session for the registrant's user account and then submit the lost password form on behalf of an account whose e-mail the attacker receives. Working backwards from the unique number assigned to the attacker's lost password request, it wouldn't take very many tries to locate the login and password change session set up on the server for the target domain. There was no security mechanism added to the lost password login process except a presumption that users could be trusted to use only the unique number assigned to them for their lost password automated login URL.

In addition to allowing flaws in Web application security to impact domain registrants' sole exclusive control over the domains they had registered, Network Solutions Internet services have been attacked at various times with varying degrees of success. Attackers who succeed even partially in attacks against parts of the Internet as crucial to its security

and function as a domain registrar end up with substantial degrees of unauthorized control over portions of the network. The attacks mounted successfully against Network Solutions include one in particular during July of 1997 where an attacker with political motives hijacked the DNS for the InterNIC Web site, managed at the time by Network Solutions. The attacker redirected large numbers of visitors to a different Web server that represented itself to be the AlterNIC where users were greeted with political propaganda against Network Solutions' government-sponsored monopoly in the domain registration business. This attack was possible because DNS server software in use at the time was designed to blindly trust as authentic any instruction received from anywhere on the network that asked the DNS server to update its cached domain to IP address mappings.

Security for TCP/IP and DNS just wasn't a priority when these protocols were first designed because only authorized, trustworthy institutions were meant to make use of them and even then only for communications that didn't require a high degree of confidentiality and protection from attackers connected to the same network. If the only way to gain access to the network is to be part of a trusted institution, and nothing of any great secrecy or importance is communicated across the network, then trusting any participant on the network to be truthful about DNS cache change requests makes a certain amount of sense. It's important to be aware that TCP/IP networks evolved out of just this type of setting where presumed trust was the rule and security the exception.

Presumed Trust in Unauthenticated Connections

The security risk of presumed trust goes beyond the reality that you can't trust your users, not even your authentic ones. There are several subtle threats that may catch you off-guard unless you carefully trace the flow of trust through the network. You may know already what this part of the book emphasizes: security for IIS is dependent on any number of external factors, and the weakest security of any of these dependencies is the effective security that your servers offer to users. But you may not have considered the multitude of scenarios in which your best security measures, including the use of Secure Sockets Layer (SSL) encryption, are already inadequate. For starters, examine Figure 8-1 which depicts a focused man in the middle attack against a particular network node. The man in the middle is depicted as positioned between the node's network interface unit and the hub or switch that the node expects to communicate with first when transmitting or receiving data.

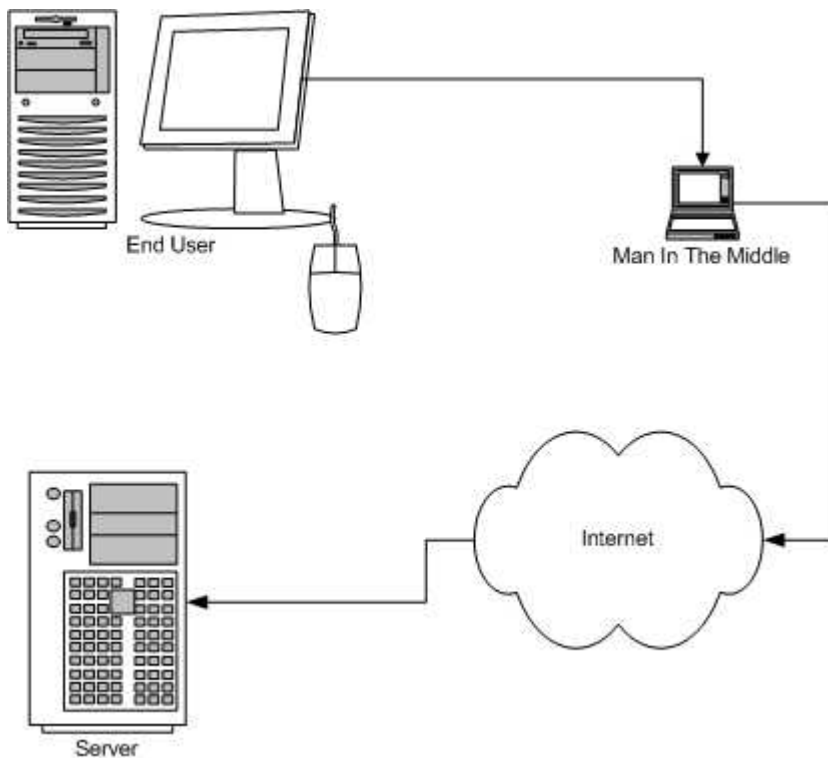


Figure 8-1: A man in the middle attack targeting a single network node

Such a man in the middle is in complete control of what the target node accesses on the real network and he can selectively alter any unencrypted data that is received by the node or impersonate the entire network by intercepting all inbound and outbound traffic. This does the attacker no good when it comes to impersonating SSL secured Web sites because the browser client possesses a root certificate from a Certificate Authority (CA) that issues the Web server's encryption certificate and the Web browser validates the Web server's certificate using its copy of the CA root certificate. A user who types an `https://` URL into the browser on the target node can be confident that they are communicating with the authentic SSL secured Web server on the real network protected by two-way encryption if the Web browser does not complain about a mismatch between the certificate offered to the client by the server and the fully qualified domain name (FQDN) of the host that the browser believes it contacted because the Web server certificate includes the FQDN of the server that the CA certified. This level of trustworthiness makes programmers happy because they can argue that even the most severe man in the middle attack is unable to circumvent the encryption, authentication, and repudiation features of SSL. But if you're an administrator you know better. End users don't type `https://` and they don't understand certificate/FQDN mismatch warning messages like the one shown in Figure 8-2 that get displayed by the browser as the only preventative measure to give the user feedback about the dire security consequences of proceeding with encrypted communications with a server that is unable to authenticate its identity.

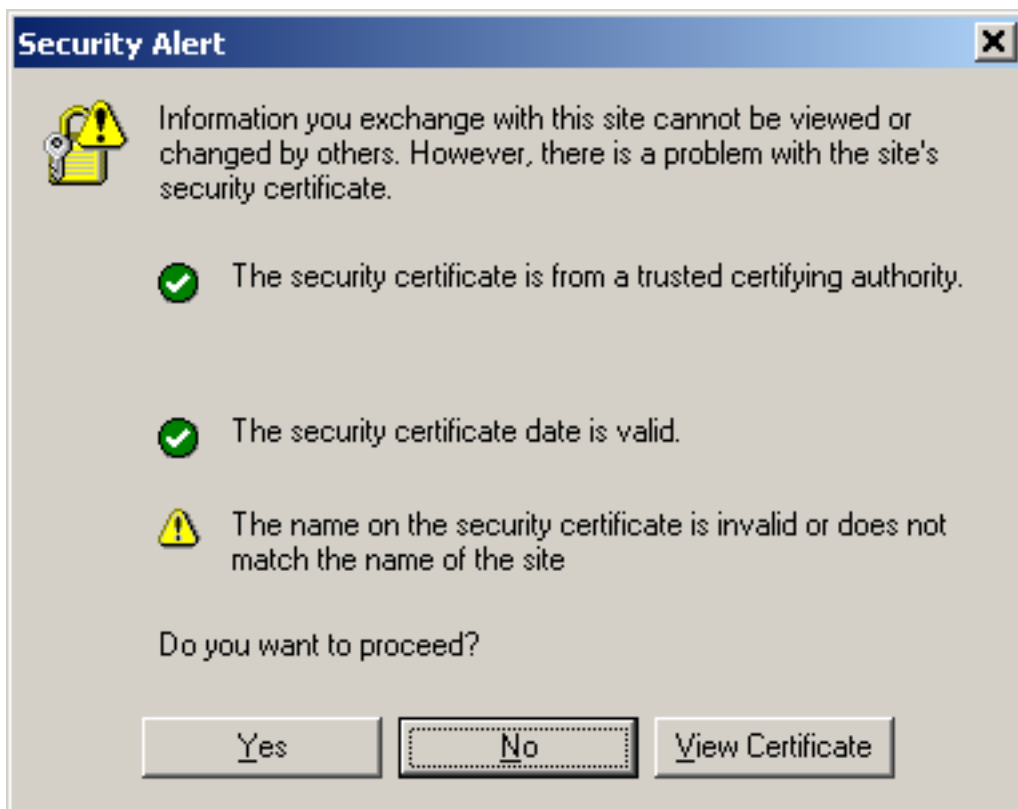


Figure 8-2: The server's FQDN is validated by the client browser to authenticate its identity

An end user who wishes to initiate an SSL encrypted connection with a Web server usually types in the `http://` URL of the server, or the server's FQDN by itself without a URL style protocol prefix, in order to make initial contact with the server. The unencrypted, unauthenticated connection made by the client with the server located at whatever IP address the DNS lookup instructed the client to use for the specified FQDN results in a Web page that the user reads to find a link to click to switch to an SSL secured connection with a server that is authorized to conduct SSL secured communications with the user. The end user won't have any special expectation of the `https://` URL to which their browser is pointed after they click the link for connecting to the authorized secure server. Therefore any man in the middle attack can successfully impersonate your SSL secured Web server without producing a security alert message like the one shown in Figure 8-2 simply by directing the user to a different SSL secured server.

The attacker preserves the user interface that the user expects to see on your site and takes advantage of the fact that the user doesn't know what the `https://` URL was supposed to have been so they have no clue that they are communicating with the wrong server. This type of attack is detectable only by a user who has intimate technical knowledge of the design and FQDNs of the Web servers that are authorized to serve a particular Web site. Even a paranoid and alert power user won't stop using an SSL secured Web site just because the FQDN portion of the URL changes when the user switches from the `http://` address to the `https://` address unless the user knows what FQDN to expect for the SSL secured server. A user who types `https://` into the browser address line without making a transitional visit to an unauthenticated, untrustworthy `http://` URL is the only user who can avoid falling victim to a man in the middle attack.

A security mechanism that hinges on the end user's ability to identify suspicious FQDNs in https:// URLs embedded in Web pages that somebody else created is adequate security only for extremely cautious programmers and administrators who access the Web sites they personally build and maintain. It is painfully inadequate security for end users. End users are protected more by the fact that nobody cares about them enough to bother mounting a man in the middle attack against their network node than they are protected by encryption and authentication facilities of SSL. Large scale man in the middle attacks against entire networks are of much greater concern, and the only protections anyone has against such attacks are:

Personal knowledge of the https:// URL to which they wish to connect

Personal knowledge of the Certificate Authority (CA) that issued the only authentic SSL certificate for the authentic server's FQDN common name

Server authentication performed by the client browser using a CA root certificate that depends on the CA protecting its secret key from becoming compromised

Awareness to look for the 12 by 15 pixel padlock icon that the browser displays along the bottom of the browser window when an SSL connection is established with a server whose identity has been authenticated

Awareness and willingness to take the time to examine the server's encryption certificate when the SSL connection is initially established

These protections also hinge on trust. Trust that the server operator is able to prevent a third party from obtaining the server's secret key for its associated certificate. Trust that the secret key used by the CA to generate Web server certificates will never be compromised due to a security breach or a successful cryptographic hack. Trust that the source of the instruction to visit a particular https:// URL was authentic. Trust that the server operator is able to adequately secure the server from unauthorized changes in its programming.

One technique an attacker can use to take advantage of users' misplaced trust is for the attacker to hijack DNS of your site such that the DNS hijacking impacts end users' name resolution of your site's domain. The Attacker is able to direct users to a masquerading malicious server which then functions as an unauthorized proxy, or an unauthorized man in the middle (MITM), that mediates all access to your site by the hijacked user. A new standard for secure DNS, DNSSEC, will make it harder to hijack DNS servers for any domain when the new standard is deployed widely by defeating attacks that occur downstream from the authoritative nameservers for the domain as registered with the domain registrar. DNSSEC defines new zone records for each domain to distribute public keys and signed hash codes that allow downstream non-authoritative DNS servers to validate the data sent to them from other DNS servers upstream.

For DNSSEC details see <http://www.ietf.org/html.charters/dnsexp-charter.html>

Users can type in the https URL to go directly to your site using SSL for both encryption and authentication of your server's identity if your site supports SSL and the end user has personal knowledge of the correct FQDN for your site's SSL server. In this case the MITM attack fails, even though the MITM is still in the middle, because the MITM is unable to use break the SSL encryption. If, however, the user types in the http URL of your site and has no personal knowledge of the FQDN of your site's SSL server then a MITM attacker is able to fool your user into clicking on a hyperlink to a malicious SSL

server, which obviously uses an FQDN other than your site's authentic SSL server but the user sees exactly what they expect to see in the way of a user interface and Web application services because the malicious server makes an SSL connection to your authentic SSL server and plays the SSL-secured man in the middle.

SSL defeats a man in the middle attack even though the MITM is still in the middle by virtue of DNS hijacking only if the end user types the https URL explicitly as the address of the server to contact. The MITM has no choice but to relay SSL packets unmodified to your authentic SSL server because only your server has a copy of the secret key that corresponds to the SSL certificate issued to your server's specific FQDN therefore only your SSL server can satisfy the end user's SSL-enabled Web client and authenticate as your site's SSL server and particular FQDN. The unauthorized man in the middle still proxies end users' access to your secure server, but without the ability to eavesdrop or masquerade so there is no damage done. You've probably never seen any Web site that uses SSL caution you strongly against ever visiting the site again using an http URL, but the truth is that every request sent to an HTTP server that does not use SSL could be going anywhere and you must not trust anything the server returns, including links to SSL-secured https URLs that the unauthenticated, unencrypted Web page might provide for your convenience.

Source Address Spoofing and Routing Restrictions

Another example of presumed trust is a common flaw in older TCP/IP routers that caused them to be too permissive in the packets they allowed to pass from one interface to another through the route table. Recall that every IP packet has a source address and a destination address. Routers examine the destination address to determine where to route each packet by matching the address against the route table and subnet masks. Simplistic routers incorrectly ignore the source address, assuming it to be valid by implicitly trusting the device that sent the packet to be honest about its source address. A device that sends packets with forged or invalid source address is said to be conducting source address spoofing. The attacks possible against networks that are too permissive with respect to routing spoofed source addresses range from Denial of Service (DoS) to remote control of boxes that inappropriately use IP packet source address as authentication.

RFC 2267 details the threat of IP Source Address Spoofing and known countermeasures for network administrators. See <http://www.ietf.org/rfc/rfc2267.txt>

The minimum level of trust required for conducting secure transactions over the network with third parties is achieved through consistent application of your own security policy and the only man in the middle protection that exists to benefit typical end users: SSL. As a programmer or administrator you know that you can trust yourself and you choose to trust your server because of your ability to implement a robust security policy and personally monitor the server to detect signs of compromised security. You choose to trust others to varying degrees depending on who they are and what you have to lose if something goes wrong. You know the network can't be trusted and you plan accordingly, but to achieve the highest possible level of security you must also be able to trust the client.

Developing Trustworthy Clients

A Web browser is an untrustworthy client in the hands of an untrustworthy non-technical user.

There is nothing programmers can do to compensate for this fact except design server applications to eliminate intermediate unencrypted and unauthenticated hops in the path a user follows to access the secure server. Users must point client browsers directly to the https:// URL of the secure server to avoid deception by man in the middle or DNS hijacking attacks. Programmers can encourage users to develop this habit by making it impossible to click a hyperlink to switch from an http:// URL to an https:// URL. Administrators can train users to look for a particular FQDN in the https:// URL and read the Web server certificate to verify the FQDN after the SSL connection is established but before sending any sensitive data to the server. However, in the end there is no way to prevent malicious third parties from giving conflicting instructions, or no instructions, to users who have not received authentic instructions previously. Making your Web sites harder to use in order to make them more secure can be an unacceptable trade off and it is no solution for the problem of presumed trust in the minds of users.

When there is no end user, security can be increased significantly. An HTTP client other than a Web browser, controlled by code rather than an end user, can be programmed to require optimal data security and reliable authentication. This means that security for XML Web services built around the IIS platform can be increased to the point that only three practical vulnerabilities exist:

1. Code on the client and server can be tampered with by an attacker or malicious code can be deployed in the form of Trojans or malicious programs.
2. The secret key used by any trusted Certificate Authority to digitally sign certificates can be compromised by theft or new cryptanalysis technique.
3. The secret key used by the client or server can be intercepted or the ciphertext produced through symmetric- or asymmetric-key encryption can be deciphered through successful cryptanalysis.

If you don't wish to trust a third party Certificate Authority then you can be your own CA by running your own Certificate Server and deploying your CA root certificate to each of your end users as shown in Chapter 14. Do this if your application security and client (user) base demand optimal security and you believe you can do a superior job of protecting your CA root certificate's corresponding secret key. Issue client certificates to each user and don't allow another form of authentication from clients to IIS other than client certificates. By being your own CA you avoid the annual fees required by a commercial CA for the certificate services they provide. This step should be an all-or-nothing switch to your CA that includes deleting all other root CA certificates in the client's root certificate list, otherwise an attacker can potentially circumvent your CA by obtaining a certificate from a third party CA that is also trusted by the clients that access your SSL-secured IIS boxes. Chapter 14 explains certificates and being your own root CA in more detail.

DNS Spoofing and Hijacking Attacks

By design, any DNS server that is unable to provide an answer to a DNS lookup request defers to another DNS server for an answer. It is common for DNS servers to recursively defer to other DNS servers until one of them provides an answer to the request and the recursion ends. Each DNS server involved in a recursive lookup caches the result for a

period of time so as not to bother the other DNS servers again unnecessarily and consume network capacity. Even in cases where network capacity consumption is not a concern, performance is increased for applications when DNS servers can avoid recursive processing of lookup requests. Any DNS server that another DNS server relies on for recursive processing of lookup requests can provide an invalid response and all downstream subordinate DNS servers involved in the recursive lookup will be poisoned with bad cache. Intentional DNS cache poisoning is referred to as DNS spoofing.

A DNS hijacking attack against an IIS box diverts users to a third party Web server by altering the IP address that one or more DNS servers return in response to name resolution requests, possibly through DNS spoofing. An authoritative DNS hijacking is one where the domain's authoritative DNS servers are compromised by the attacker, giving full and immediate control over DNS lookup results received by any network node that relies on a subordinate DNS server without cached lookups or access to a recursive DNS resolver with cached lookups that occurred prior to the hijacking. Authoritative hijacking eventually results in complete control of all DNS lookups as cache is flushed by DNS servers around the Internet and the attacker's replacement addresses take over.

Subordinate DNS hijacking is any hijacking of subordinate DNS servers where the number of users impacted by the hijacking is equal to the number of users whose network nodes rely on the hijacked subordinate nameservers.

Authoritative hijacking can be combined with hijacking of the subordinate DNS servers that service the domain registrant's own network to delay detection of the hijacking by the domain registrant. As with any DNS attack, the authentic servers can either be penetrated, poisoned or spoofed. The difficulty of penetration may necessitate spoofing as the only attack option. If penetration is possible, DNS hijacking is a simple matter of modifying zone files to change the DNS server configuration. Replacing DNS server software with a Trojan is an option if the attacker can alter the system's software or install new software.

An attacker incapable of penetration may still be able to mount a physical attack on network infrastructure to divert network traffic to a box that spoofs the hijacked server. Or the attacker can poison the cache of a DNS server by spoofing upstream DNS servers or modifying packets that contain DNS response data through a packet sniffer attack. The subordinate DNS servers used by the registrant for name resolution can be hijacked so they are no longer properly subordinate to the authentic authoritative servers in order to give the registrant the appearance that DNS for the domain is functioning normally while authoritative hijacking is in progress. In short, there are more ways for attackers to mess up DNS than there are ways for it to work properly, and there's nothing you can do to secure subordinate DNS servers that don't belong to you.

Vulnerabilities Produced by Insecure DNS

DNS hijacking can be used to mount three different types of attack. The first attack is a simple hijacking and credential capture attack, where the third party Web server fools the user into supplying credentials by serving up a copy of the authentic server's HTML and client-side scripts. After the user's credentials are captured, the third party server may display an error message indicating that the server can't complete the requested

operation due to a temporary problem, planned service outage for maintenance purposes, or some other excuse that is meant to convince the user to try again later. The symptoms of this type of attack from the perspective of IIS are limited. IIS won't see any client requests from hijacked users, so the scale of the attack has to be very large in order to produce a noticeable decrease in client requests. Contact from users who are unable to login successfully to your server may also be a resulting symptom.

The second attack is a man in the middle (MITM) attack whereby the third party server relays to the authentic server every request it intercepts from clients in order to receive from the authentic server an authentic response that it can deliver to the client. This type of attack is extremely difficult or impossible for the end user to detect without personal knowledge of the FQDN of the authentic SSL-secured server or the IP address of the authentic server if SSL is not used. However, this attack is relatively easy to detect from within IIS as automated countermeasures are possible.

The third attack is a trust attack that convinces users to give the attacker sensitive personal information such as credit card or bank information by taking advantage of the trust a user feels for the organization that controls the authentic server. This type of attack works even when the authentic server doesn't offer any type of e-commerce or user account facilities because users are gullible, especially when they are intoxicated by trust. The attacker has little trouble convincing users that the authentic organization is offering something new to its Web site users that they qualify for only by providing sensitive personal information or payment. Every user feels as though a trusted organization may legitimately offer something in the way of e-commerce to its Web site users, so there's virtually nothing that can be done to protect against this type of attack unless users are trained to look for and trust only a particular FQDN when secured and authenticated via SSL.

Preventing Hijacking Attacks

One of the reasons an attacker mounts a hijacking attack is to capture user credentials in order to offer false client authentication to the server in the future to gain access to privileged resources. The simplest hijacking attack is one that produces a false negative authentication response rejecting the user's credentials and pointing the finger of blame back at them. An example of this type of attack in everyday life is the construction of a fake bank automated teller machine (ATM) that allows ATM card users to attempt cash withdrawals and other banking transactions. After the victim has typed in the PIN number for the card, the fake ATM rejects the card and instructs the victim to contact their bank for assistance. The attacker forges a reproduction of the card's magnetic stripe using the information read from the card when the victim inserts it into the fake bank machine and uses the forged card and the intercepted PIN number credentials to withdraw cash from a real bank machine. The fake ATM machine isn't a hypothetical attack – it has actually occurred in the real world and probably will again. The only defense banks implement to protect account holders from such spoofing and hijacking credential capture attacks is to remain aware of the locations of their authentic ATM machines and publish those location lists for the benefit of security-conscious customers.

Credential capture hijacking attacks are especially difficult to defend against because the attacker mounts the attack simply by impersonating, or spoofing, your equipment in such

a way and in such a location as to fool the user. The only defense possible against an attack that occurs completely outside the equipment and physical space over which you have control and that takes advantage of end users who are ill-equipped to detect such an attack themselves is to better equip your end users. You can design your IIS applications to make it impractical for attackers to mount credential capture attacks by implementing the following user login procedure:

1. Prompt the user first for user ID only
2. Display a challenge phrase known to the user
3. Ask the user to verify the challenge phrase and click Yes if it's authentic
4. Prompt the user to enter their password

This four step procedure forces an attacker to mount a man in the middle attack rather than mounting the simpler, and more easily concealed, credential capture attack. Code shown in the next section automatically detects and disables man in the middle attacks, allowing damage control including automatically disabling credentials captured by the attacker. Other hijacking and credential capture countermeasures are possible, and the more specific to your application and more custom-tailored to your user community the better. One of the best ways to equip your users with a hijacking and credential capture countermeasure is to issue them a list of challenge phrases that they are instructed to expect from the authentic server. Each time the user authenticates with the server one of the challenge phrases is used and crossed off of the list. When the list is used up, the user receives a new list. This restricts the attacker's options for mounting attacks by forcing the attacker to intercept the list. This type of one-time-use list is compatible with every user and every Web browser because of its simplicity. IIS is simply programmed with application-specific logic to display the challenge phrase to the user before the user provides credentials.

The goal of such an application-specific hijacking countermeasure is to make sure that the attacker is dependent on your server to convince the end user that their trust is not being misplaced. That is, to guarantee for the end user that even if there is a man in the middle acting as an unauthorized proxy between them and the authentic server, at the very least the end user can be certain they are in fact communicating with a device on the network that is in turn communicating with the authentic server. This way the authentic server can implement automated countermeasures to defend itself and its users against man in the middle attacks. A one-time-use list deployed only for server authentication provides no protection against an attacker providing false client authentication using captured credentials. To protect against credential capture, a second one-time-use list of matching response passwords can be given to each user along with the list of challenge phrases.

Such a one-time-use password list limits the damage an attacker can do to only those services the server was going to provide to the authentic user during the authenticated session. The attacker is unable to authenticate in the future using the captured password, and any attempt to do so can be viewed by the server as evidence that the transactions performed during the previous session where that particular password was supplied may have been compromised or initiated by an attacker with the captured password. For optimal authentication of both client and server, one-time-use list pairs can be issued for each distinct function provided by the server so that an attacker who gains control over an authenticated user session is unable to execute commands on the server that are

outside the scope of the function being performed by the authentic user. The server authentication provided by SSL is good, provided that users have first-hand knowledge of the authentic FQDN to which they are supposed to connect and assuming that none of the Certificate Authorities whose root certificates are trusted by Web clients lose exclusive control over their secret keys. Client certificates are also good, but both can be stolen or compromised in various ways. Whether or not SSL is used in your application, application-specific mutual authentication is still an important part of data security as one of the only defenses against outright hijacking and spoofing.

Of course, end users who aren't trained to expect a challenge phrase during login will fall victim to the malicious server that simply leaves that part out and asks for both user ID and password in a single step as users are conditioned to expect at most sites. Therefore, another important consideration is to plan in advance for credential capture hijacking attacks to occur and build into your Web application a way to determine over the telephone with an affected user that a hijacker diverted the user to another server. It's impossible for your technical support staff to assist the user if the user's Web client is contacting a third party server instead of your authentic server, and technical support inquiries are an important line of defense as a source of information about attacks against your servers.

One way to increase the counter-intelligence gathering value of end user technical support calls is to assign each HTTP request a unique transaction stamp and send it to the client as part of the HTTP response. Then, when a user contacts your technical support line asking for help logging into the server, verify that the response received by the user's Web browser contains a valid transaction stamp. If not, you've detected an attack that is diverting users to a third party server and can mount an appropriate defense response. Including the client IP address from which the server received the HTTP request as part of the transaction stamp is important so that it can be compared with the end user's IP address when obtained by tech support. The absence of a transaction stamp confirms that the user was hijacked while a mismatch between the IP address of the user's computer (or authorized proxy server) and the IP address contained within the transaction stamp confirms the presence of an unauthorized proxy server man in the middle.

The following code, written in C# for ASP.NET, shows how to create a response filter stream that modifies the response sent by IIS to any request for .aspx files. This particular filter doesn't attempt to make the transaction stamp visible to the end user, so the user will have to view the HTML source of the Web page they've accessed in order to convey the contents of the transaction stamp to you or your technical support staff. The code as shown inserts a <META> tag at the top of the response body, which will normally place it outside the <HTML> open tag. Ideally <META> tags appear within the <HEAD> section of the HTML document, but in this case, for simplicity and because the transaction stamp isn't meaningful to the browser or any search engines and its non-standard placement shouldn't impact the functionality of either type of client, the code just inserts the <META> tag first.

```
using System;  
using System.Web;  
using System.Security.Cryptography;
```

```

using System.IO;
namespace TransactionStamp {
public class TransactionStampModule : System.Web.IHttpModule {
Rijndael aes = null;
ICryptoTransform aesEncryptor = null;
String sAppendToLog = null;
public void Init(HttpApplication context) {
context.BeginRequest += new EventHandler(this.TransactionStamp);
aes = Rijndael.Create();
aesEncryptor = aes.CreateEncryptor();
sAppendToLog = "TransactionStamp > " +
DateTime.Now.Ticks.ToString() + ": Key=" +
BitConverter.ToString(aes.Key) + " IV=" +
BitConverter.ToString(aes.IV) + "\r\n";
public void Dispose() {}
public void TransactionStamp(object sender,EventArgs e) {
HttpApplication app = (HttpApplication)sender;
String stamp = null;
String timestamp = null;
byte[] plaintext = null;
try { if(sAppendToLog != null) {
app.Response.AppendToLog(sAppendToLog);
sAppendToLog = null;}
if(app.Request.FilePath.EndsWith(".aspx")) {
timestamp = DateTime.Now.Ticks.ToString();
stamp = timestamp + ":" + app.Request.UserHostAddress;
plaintext = System.Text.Encoding.ASCII.GetBytes(stamp);
stamp = timestamp + ":" + BitConverter.ToString(
aesEncryptor.TransformFinalBlock(plaintext,0,plaintext.Length));
stamp = "<META name=\"TransactionStamp\" content=\"" +
stamp + "\">\r\n";
app.Response.Filter = new
TransactionStampFilter(app.Response.Filter,
System.Text.Encoding.ASCII.GetBytes(stamp)); } }
catch(Exception ex) {}
}}
public class TransactionStampFilter : Stream {
private Stream streamFiltered = null;
private byte[] stamp = null;
private bool stamped = false;
public TransactionStampFilter(Stream s, byte[] b) {
streamFiltered = s;
stamp = b; }
public override bool CanRead {
get { return streamFiltered.CanRead; }}
public override bool CanSeek {
get { return streamFiltered.CanSeek; }}
public override bool CanWrite {
get { return streamFiltered.CanWrite; }}
}
}

```

```

public override long Length {
get { return streamFiltered.Length; }}
public override long Position {
get { return streamFiltered.Position; }
set { streamFiltered.Position = value; }}
public override long Seek(long offset, SeekOrigin dir) {
return streamFiltered.Seek(offset, dir);}
public override void SetLength(long len) { streamFiltered.SetLength(len);}
public override void Close() {
streamFiltered.Close();}
public override void Flush() {
streamFiltered.Flush();}
public override int Read(byte[] buf, int offset, int count) {
return streamFiltered.Read(buf, offset, count);}
public override void Write(byte[] buf, int offset, int count) {
byte[] newbuf = null;
if(stamped) { streamFiltered.Write(buf, offset, count); }
else {newbuf = new byte[count + stamp.Length];
Buffer.BlockCopy(stamp, 0, newbuf, 0, stamp.Length);
Buffer.BlockCopy(buf, offset, newbuf, stamp.Length, count);
streamFiltered.Write(newbuf, 0, newbuf.Length);
}}}}

```

The code shown applies simple encryption to transaction stamp every response with a string that combines the client's IP address with the current date and time on the server. The ciphertext output by the encryption algorithm is converted from an array of bytes to a sequence of hexadecimal values more easily communicated by an end user. The TransactionStampFilter class inherits from System.IO.Stream and its constructor accepts two parameters. The first parameter is the Stream object to filter, or wrap, with the instance of TransactionStampFilter. The value passed in for this parameter is stored in a private variable named streamFiltered for future use by the TransactionStampFilter Stream object each time data is written to the client by the ASP.NET application. The second constructor parameter is the byte array containing the transaction stamp META tag.

When you build the code using the C# compiler, you will typically assign the code a strong name using the .NET Strong Name Utility (SN.EXE) and Assembly Linker (AL.EXE) and you must also deploy the IHttpModule to an ASP.NET-enabled server by modifying the machine.config file in the Microsoft.NET CONFIGURATION directory. Add the namespace and class name for the TransactionStampModule to the list of HttpModules as shown:

```

<httpModules>
<add name="TransactionStamp" type="TransactionStamp.TransactionStampModule,
TransactionStamp"/>
</httpModules>

```

A simple transaction stamp will distinguish a page produced by your authentic IIS box from one produced by an attacker's malicious masquerading server because the attacker's

box will be unable to produce a transaction stamp that will decrypt using your server's secret encryption key without contacting your authentic IIS box and making a request – and doing so in order to send a valid transaction stamp to the end user will reveal the attacker's IP address. When the attacker is forced to contact your authentic server, the nature of the attack changes from one of intercepting and diverting users, a user capture hijacking attack, into one that can be characterized as a man in the middle attack. Because there are conceivable cryptanalysis attacks that may be able, given enough computing power and time, to discover the encryption key and initialization vector used in the encryption and decryption performed by the code shown, changing the encryption key and IV periodically eliminates even this minor concern if necessary in your real world usage scenario.

Detecting and Preventing Man in the Middle Attacks

Every man in the middle must be considered an attacker or else every man in the middle attack succeeds. Designing applications around IIS that include some common sense awareness of the man in the middle threat is relatively simple. Automated prevention or at least damage control is also feasible as described in this section. The most difficult of all man in the middle attacks to detect is one that targets only a single network node as depicted back in Figure X because the man in the middle has every appearance from IIS' perspective of being the actual client network node. By comparison attacks that target many nodes at once or in series are easier to detect. There are various symptoms that suggest a man in the middle attack may be in progress. Symptoms that automated defense mechanisms for IIS can reliably use as lockdown triggers are the most important to understand.

There are three types of man in the middle attack as viewed from the perspective of an IIS box: packet sniffing with inline tampering, TCP hijacking, and spoofing or masquerading as the IIS box using a third IP address. Packet sniffers monitor IP traffic and modify packets in order to alter the substance of communications between the client and server. In the most severe packet sniffing attack, all packets are captured by the man in the middle and are regenerated selectively or absorbed completely. The man in the middle can forge packets in both directions and neither end of the communication is able to detect the attack because packets that do reach them are addressed and formatted exactly as they would have been if they had not been forged or regenerated by the man in the middle.

TCP hijacking reroutes packets to the attacker's box that otherwise would have reached the intended recipient. The attacker doesn't modify packets, instead packets are redirected at the level of a router, switch or hub. One of the known TCP hijacking attacks involves poisoning the Media Access Control address (MAC) to IP address table maintained by conventional Ethernet hubs using the Address Resolution Protocol (ARP). The ARP table keeps track of the physical devices that are directly connected to the local area network so that the proper MAC address can be used to send data to the device that is believed to be using the IP address that appears in the destination address field of an IP packet. When the attacker wants to hijack packets the ARP table of the hub is modified so that the MAC address of the attacker's box, which is also connected to the hub, replaces the MAC address of the authentic box. This causes Ethernet frames to be addressed

improperly to the attacker's box when those frames carry IP traffic addressed to the hijacked IP address.

The poisoning is reversed on-demand to resume Ethernet transmissions to the authentic box. IIS see no symptoms of this attack when executed against the client but experiences intermittent network communication failures when it is the target of the attack.

Spoofing or masquerading is accomplished with a box that is configured to impersonate the authentic IIS box. DNS hijacking is one way for an attacker to mount a spoofing attack and it is by far the easiest method because it doesn't involve tampering with routers or physically connecting to other people's hubs and switches, it simply takes advantage of the trust built into insecure DNS. TCP hijacking can be combined with spoofing, and must be if the TCP hijacking attack is directed at the server rather than the client. For the attack to be classified as a man in the middle attack, the server that masquerades as the authentic IIS box must do more than just capture client connections and service them as in the credential capture attack, it must act as an unauthorized middle man to receive connections from clients and initiate corresponding connections with the authentic server in order to capture all data exchanged and provide the client with the application functionality and visual appearance produced by the authentic server.

Automated Detection of Proxies and a Deny-First Countermeasure

To the authentic IIS box, the masquerading/spoofing man in the middle appears very similar to a proxy/firewall or DHCP address pool through which multiple clients access the Internet. The man in the middle uses a single IP address or a small address pool to initiate requests to the authentic server that correspond to each client request it intercepts just like a proxy/firewall does on behalf of clients on a protected internal network. You know that a spoofing man in the middle attack may be occurring when authentication credentials for more than one user account are provided from the same IP address. The only legitimate explanation for this is that all of the authentic users whose credentials were relayed to your IIS box from the same IP address rely on the same proxy/firewall or DHCP server to access the Internet. In this case the man in the middle is authorized on behalf of the users to function as a man in the middle and SSL-secured connections are trustworthy because the man in the middle simply relays encrypted data to and from the clients it services. A proxy can't decrypt the data sent by either side of an SSL connection the way that an unauthorized man in the middle can.

An unauthorized man in the middle can redirect clients to its own replacement SSL-secured server to take advantage of the fact that users lack personal knowledge of the FQDN to expect for the authentic SSL-secured server. The attacker can receive and service SSL-secured connections from clients by originating SSL-secured connections to the authentic SSL-secured server in order to provide users with the application services and user interface they expect of the authentic secure server. In doing so the unauthorized man in the middle completely bypasses SSL data security for privacy and the only symptom visible to the client is hidden behind the technical and logical complexity of reading client certificates, suspicious FQDN common names and the user's ability to subjectively perceive something as being suspicious. Consider an attacker who controls the domain `express.com` who obtains an SSL certificate for `american.express.com`; how many users

will fail to perceive this FQDN as suspicious when they contact americanexpress.com and a man in the middle alters the address of the SSL-secured server to which users are directed when they login?

DNS Pooling as an Automated Countermeasure Against DNS Attacks

One technique for automatically detecting a man in the middle attack mounted against an IIS box by way of DNS hijacking or spoofing is to force the client to send requests to multiple domains in order to enable the server to compare and match the remote IP address of each client connection. If the client's IP address is the same in every request, then your IIS box is able to deduce that DNS resolution for the domains has not been poisoned or hijacked from the perspective of that particular client. The key feature of this countermeasure is utilizing a pool of domains and obtaining DNS hosting from a different ISP for each domain. To bypass this security countermeasure, a malicious man in the middle has to filter every response from IIS to modify absolute URLs that don't reference the hijacked domain – DNS hijacking is no longer a viable option as a means to insert a MITM undetected by IIS. The attacker will then have to act as a man in the middle for those requests, too, and perform address translation to map the modified URLs back to the originals for requests it initiates to those servers in order to retrieve content that the user expects. Even if the MITM hijacks every domain in the domain pool, IIS are still able to detect the attack and automatically lock out the malicious proxy server by IP address.

This countermeasure works because of the way that changes to DNS propagate out in waves from the authoritative nameservers to subordinate DNS servers that only cache lookup results. Provided that the domains in the domain pool are used by end users to access live Web sites, and provided that there is enough traffic to those domains, each request made by an end user for content from a server located at one of the domains creates a ping of the DNS health of the servers at the other domains. If a man in the middle hijacks one of the domains and redirects requests for content from that domain to a malicious proxy server, the authentic server will be able to detect that the IP address for the request relayed by the proxy server is different than the IP address used by the authentic client to make the other requests to the other domains that have not been hijacked. There are several variations to this defense, and they all require the server to keep track of and match client IP addresses for requests across multiple domains. The simplest variation of this defense uses only two domains and a simple dynamic URL that is generated by the server and given to the client to request. The client can be given the dynamic URL as a hyperlink, as part of an tag, or as a frame in a frameset. Whatever makes the most sense for the application. The dynamic URL points at the second domain, not the one that the Web browser was directed to in its request for the page.

A dynamic URL is necessary to decrease the likelihood that an authorized friendly proxy server between the client and server will serve the URL out of cache when the client requests it. To simplify deployment of this defense, the server that generates the dynamic URL can embed the client's IP address, or an encoded form of it, in the URL path. The server must also process the response dynamically so that it can parse out the encoded IP address and match it against the IP address from which the request arrived at the server. If a malicious MITM relays the request for the dynamic URL from the client, the MITM will expose its IP address to IIS. The simple fact that the client IP address is

different in the request to the dynamic URL than it was in the request to the initial URL is proof that two network nodes were involved in making the request rather than just one.

The following code demonstrates a dynamic URL generator written in C# for use in ASP.NET, implemented as a static class method, and its corresponding parser module that singles out MITM attackers by detecting multiple IP address mismatches originating from the same IP address. To keep the code simple so that it makes sense easily, the dynamic URL is constructed by appending the client's IP address to "http://FQDN/dynamic/" exclusive of the dots (periods) in the dotted quad IP address and giving the resulting URL a .jpg file extension. Replace FQDN with a fully qualified domain name by which your IIS box is also accessible, but be sure to use a domain other than the one that serves the Web application. Do the same thing in reverse to deploy this same countermeasure as part of a Web application served from the FQDN so that the two domains depend on each other for DNS hijacking MITM automated detection.

This design variation is only the simplest of the potential embodiments of this automated DNS hijacking MITM countermeasure. It doesn't provide a defense against a MITM that serves requests for .jpg files out of cache instead of relaying them on to the authentic HTTP server, but it demonstrates the countermeasure concept so that you can enhance the defense with your own server-side security policy. This minimal countermeasure design, as shown, does provide a defense against a MITM attack that has not itself been designed to detect and circumvent this variation of the countermeasure by selectively relaying requests from the client to the authentic server and cleansing client requests of any identifying dynamic elements. To eliminate all possible attacker circumvention of this automated DNS hijacking countermeasure requires a more complex server-side awareness of the countermeasure and its role in securing client sessions, including a security policy that says the server will not allow a client session to proceed until the dynamic URL issued to the client gets requested. In this case the only way for the attacker to circumvent this countermeasure is to hide, which means the attacker is denied the MITM scenario.

```
using System;
using System.Web;
using System.Security.Cryptography;
using System.IO;
using System.Collections;
namespace MITMCounterMeasure {
public class MITMCounterMeasureModule : System.Web.IHttpModule {
    SortedList KnownClients = new SortedList(131072);
    SortedList MITMSuspects = new SortedList(10240);
    public void Init(HttpApplication context) {
        context.AuthorizeRequest += new EventHandler(this.MITMDetect); }
    public void Dispose() {}
    public static String GenerateDynamicURL(String ClientIPAddress) {
        String sURL = "http://FQDN/dynamic/";
        sURL = sURL + ClientIPAddress.Replace(".",null) + ".jpg";
        return(sURL); }
    public void MITMDetect(object sender,EventArgs e) {
        HttpApplication app = (HttpApplication)sender;
```

```

try { String addr = app.Request.UserHostAddress;
addr = addr.Replace(".",null);
if(MITMSuspects.Contains(addr)) {
throw(new Exception("MITM Suspect: Request Denied")); }
String urlpath = app.Request.Url.AbsolutePath;
int i = urlpath.IndexOf("/dynamic/");
if(i > -1) {
i = i + 9;
String s = urlpath.Substring(i,urlpath.IndexOf(".")-i);
String v = null;
if(!s.Equals(addr)) {
if(KnownClients.Contains(addr)) {
v = (String)KnownClients.GetByIndex(KnownClients.IndexOfKey(addr));
if(!v.Equals(s)) {
SortedList.Synchronized(MITMSuspects).Add(addr,DateTime.Now);
throw(new Exception("Second IP address mismatch different from first mismatch: MITM
suspect identified"));
}}
else {
SortedList.Synchronized(KnownClients).Add(addr,s);
}}
// send data to client; in this example the response is a JPEG image
app.Response.WriteFile(@"\inetpub\wwwroot\image.jpg");
app.CompleteRequest();
}}
catch(Exception ex) {
app.Response.Write("<html><body><h1>Error Processing Request</h1></body></html>");
app.CompleteRequest(); }
}}}

```

The code shown verifies that the client IP address matches the address encoded in the URL as an unauthorized proxy server detection countermeasure. This is the simplest of the possible multiple-domain anti-DNS-poisoning countermeasures enabled by virtue of the fact that changes to DNS cache always propagate unevenly from different authoritative DNS servers for different domains. Even if a DNS hijacker takes over the authoritative DNS servers for each domain used to accomplish this countermeasure, the attacker is unable to guarantee that the DNS poisoning will spread evenly throughout the network of subordinate DNS servers. That is, there will always be a period of time during which certain DNS servers on the network contain a mix of authentic DNS information for one or more of the hijacked domains and poisoned information for one or more of the hijacked domains. This time window represents an opportunity to detect the DNS hijacking and defend IIS against it. The amount of time in this window can be influenced, though not controlled precisely, through the Time To Live (TTL) setting in each domain's authoritative DNS server.

As discussed in Chapters 5 and 6, ASP.NET provides an improved architecture for layering- in IIS code modules that participate in HTTP request processing. ASP.NET layered code modules are meant to replace the legacy ISAPI architecture, rescue IIS developers and IIS administrators from DLL Hell, and provide a solution that eliminates the difficulty of

writing secure code in C/C++. The C# code shown in this section to implement the simplest potential DNS hijacking MITM countermeasure relies on the ASP.NET IHttpModule interface. It only works if you set up a file mapping for .jpg files in the Web site application configuration that instructs IIS to use the ASP.NET script engine to process requests for all JPEG image files. When you build the code using the C# compiler, you will typically assign the code a strong name using the .NET Strong Name Utility (SN.EXE) and Assembly Linker (AL.EXE) and you must also deploy the IHttpModule to an ASP.NET-enabled server by modifying the machine.config file in the Microsoft.NET CONFIGURATION directory. Add the namespace and class name for the MITMCounterMeasureModule to the list of HttpModules as shown:

```
<httpModules>
<add name="MITMCounterMeasure"
type="MITMCounterMeasure.MITMCounterMeasureModule, MITMCounterMeasure"/>
</httpModules>
```

A single IP address mismatch may be caused by an end user with a dynamic IP address whose IP address changes, but more than one mismatch relayed to IIS from the same IP address indicates, as shown in the parser source code, the presence of an unauthorized MITM. This means that the malicious MITM gets to hijack one client for free but the second hijacking triggers detection and the automated countermeasure. If you're worried about dialup users who use the same DHCP address pool ending up with the same IP address one after another in sequence, such that this source code would trigger a MITM defense against the second innocent end user whose dynamic IP address changed while using your server, you can increase the number of hijackings permitted per IP address or implement a more comprehensive variation of this MITM countermeasure that keeps track of client sessions and authentication events in addition to IP addresses and uses extra information gathered about the context of the client request to avoid locking out any end users inappropriately.

Preventing Denial of Service Attacks

Denial of Service (DoS) attacks vary in style and complexity but all attempt to disable IIS request processing either directly or indirectly. Some DoS attacks are mounted by an attacker in order to force software into the unusual mode where memory buffers, the process call stack, and possibly system resources are completely saturated. Under such a heavy processing load certain software bugs such as multithreaded race conditions can reveal security vulnerabilities including buffer overflows in stack and heap variables that the attacker may be able to exploit to execute arbitrary malicious code. For this reason, some DoS attacks are more dangerous to security than the mere disruption of request processing may suggest. Additionally, DoS attacks against ancillary services such as mail servers, DNS servers, and traffic monitoring and usage profiling systems can disable portions of your server security policy to purposefully render you blind while an attacker mounts some other type of attack against IIS. A DoS attack against IIS, even if it results in no compromise to security, can fill up server request and event logs such that all subsequent activity occurs with no record-keeping until more storage space is provided or logs files are moved offline to free up storage space.

A DoS attack against authoritative DNS servers is especially useful for an attacker who is mounting a DNS spoofing (cache poisoning) attack against subordinate DNS servers. By rendering the authoritative nameservers unreachable, the attacker is able to compel a subordinate DNS server to rely on a lookup result provided by a non-authoritative poisoned DNS server. If the attacker is unable to penetrate the security of the authoritative nameservers for a given domain, throwing a DoS attack at those servers may be necessary in order to execute the planned DNS hijacking. An attacker who has succeeded in poisoning cache of various DNS servers may be able to trigger a DoS condition in any poisoned DNS server in certain circumstances. One example of a DoS attack made possible through DNS poisoning was the intentional insertion of an invalid recursive CNAME record, an alias pointing one FQDN at another, where the CNAME points back at itself. This is known as a self-referential RR and the presence in DNS cache of such a record enables an attacker to create a DoS condition in certain older DNS servers simply by requesting a zone transfer for the FQDN of the self-referential RR recursive CNAME record. The DNS server goes into an infinite recursive loop attempting to resolve the CNAME alias in order to process the zone transfer request.

Protecting IIS with a Restrictive Security Policy

Although it is possible to build permissive Web applications around IIS that automatically benefit from features of the dynamic adaptive nature of TCP/IP networks that make life easier for the end user, such automatic features reduce the security of IIS because authentic users' use patterns are permissively allowed to resemble malicious attacks. For example, a dialup user with a dynamic IP address may lose connectivity in the middle of a session and reestablish Internet access with a different IP address. If you do nothing special in IIS to detect this condition, the user can simply continue using the Web application hosted by IIS as though nothing had changed because the Web browser doesn't care what IP address is used to identify the computer on which it runs. But permitting this usage scenario means that any malicious third party from any IP address can hijack any active session just by obtaining its session identifier or authentication ticket.

There are load balancing techniques possible on the client side that could trigger automated countermeasures on the server without just cause. For example, a proxy farm could easily transmit requests to a single server on behalf of a single client from more than one IP address in order to balance traffic load outbound from the ISP that serves the client across multiple Internet connections. While such a deployment may be rare, anyone who does deploy such a client-side load balancing mechanism should re-think the design. You need to stand firm on your security policy decision to block users from such sources. If a company designs its client proxy systems in such a way as to cause its authentic users to be indistinguishable from malicious users, the burden of repair must be placed back on the company that designed the proxy. The best solution available, SSL, still has flaws that need to be supplemented with automated defense countermeasures wherever possible.

To achieve your goal of increased data security, you must adopt a security policy that is restrictive with respect to such usage scenarios in order to reserve network events like a change of IP address as fatal to the security of a session. Automated countermeasures must be possible or else the network itself prevents effective countermeasures from

being possible. If your clients must access IIS from a location on the network where proxy farms are in forced usage, you can either require that SSL be used by all such clients or exclude the IP address range that such clients use for Internet access from any automated defensive countermeasures and apply more scrutiny to the activities performed by users from those address ranges. This is consistent with the best practice security policy of differentiating between users who visit your Web sites from within your own country versus users from another country. Users from any address that is known to be part of a proxy farm that alters the user's IP address with each request must be afforded less trust than users whose IP address can be counted on to remain consistent for the duration of the client's use of IIS.

TCP/IP networks are inherently transactionless, stateless, interdependent, programmable, and under distributed control by multiple administrators. Each user is further responsible for the care and feeding of the equipment they operate that is connected to the network, most of which is also programmable. Within any network comprised of programmable computers there can never be perfect data security. However, security can be achieved to a high degree of confidence, and the risks reduced to a manageable list of known vulnerabilities. SSL and other cryptographic systems based on asymmetric public key cryptography can provide an environment for safer computing within a programmable TCP/IP network but these systems including the digital signatures they facilitate only represent best-effort trust they do not represent optimal data security. To approach optimal data security in a TCP/IP network of programmable devices under distributed administrative control and arbitrary end user ownership requires every vulnerability to be carefully mapped and every possible countermeasure built and deployed. Optimal security also requires end users and client software to be programmed with an awareness of the fundamental threats to data security including presumptions of trust they make to get work done.

Chapter 9: Transaction Processing Security

One of the core features of many information systems is the ability to ensure transactional integrity through On-Line Transaction Processing (OLTP). A transaction is a unit of work that must be completed or aborted in its entirety. Partial completion of a transaction would create an unacceptable condition that would allow bad data to exist in an information system. OLTP transactions aren't the same as business transactions. Any logical unit of work that moves an information system from one self-consistent state to another can be an OLTP transaction. It's up to programmers to decide how the stages of business transactions or other computer-assisted activities translate to OLTP transactions. This design process is well understood by developers of relational database applications. However, transaction processing has nothing to do with security. Transactional applications are able to recover reliably from faults such as hardware failures or software bugs that can cause the Blue Screen of Death but fault tolerance through OLTP does not equal security.

For an operation to qualify as a transaction it must possess four properties that together are known by the acronym ACID. The letters of this acronym stand for the following concepts. Each property serves a particular purpose as summarized below.

- Atomicity to prevent non-deterministic results
- Consistency to preserve relationship integrity
- Isolation to conceal work in progress
- Durability to ensure fault tolerance

Atomicity is the property of a transaction that guarantees that it either occurs or it does not and there is no in-between during which time other transactions might corrupt its atomic quality. An operation is truly atomic when it is impossible for any other operation to occur simultaneously. Most transactions are not truly atomic since they involve more than just single machine code instructions. A microprocessor must either execute a machine code instruction or delay executing it; once execution has started there is no way for it to suspend the instruction in order to service an interrupt. Such all-or-nothing immediate instructions are classic atomic operations and transactions simply mimic this end result property through a design that precludes conflicts with other transactions by virtue of locking and other concurrency control.

Consistency means that every change to a transactional resource is a self-consistent transformation that takes the resource from an initial valid condition to a new valid condition. Neither transaction success nor failure should be capable of resulting in an invalid condition where relationship constraints are violated. Consistency means that every part of a transaction must succeed or fail together and the changes made to data storage or actions initiated by the transaction must be self-consistent when considered together as a single transaction. Consistency and isolation are similar in that you can't have one without the other. If every partial change proposed by a participant in a pending transaction is immediately visible to all other transactions, this lack of isolation could result in transactions that make use of data that is never committed to durable storage when pending transactions are aborted.

Isolation failures can result in undetected data corruption phenomena such as dirty writes, dirty reads, nonrepeatable reads, and phantoms. A dirty write occurs when two transactions read data at nearly the same time such that they both start with the same data and both transactions modify or delete the data then commit their respective pending changes. Although both write operations succeed, the transaction that commits last is the only modification that persists, resulting in a complete loss of the other transaction's data. A dirty read occurs when data is added or updated as part of a pending transaction and the addition or the update becomes visible before the transaction commits. Any software that reads data modified by the pending transaction is liable to malfunction unless it is explicitly designed to tolerate, and compensate for, the uncertainty produced by dirty reads. Until a transaction commits, the updates that are pending in the transaction may be incomplete such that relationships between data tables and other aspects of persistent state required to maintain consistency have not yet been fully established, and the pending transaction may be aborted if one of the transaction participants fails to accomplish its appointed task.

When software accepts a dirty read condition, it must be prepared for the possibility that some or all of the data it reads may never be committed to durable storage if the pending transaction aborts and its pending changes are rolled back. A non-repeatable read occurs when a transaction updates or deletes data that has been read by another transaction. The transaction that performed the non-repeatable read will receive different data the next time it performs the same read. Phantoms are similar to non-repeatable reads but they pertain not to individual rows so much as to result sets returned in response to particular queries.

As other processes modify and delete data concurrently with a process that queries against the same data, any query condition is likely to result in different row sets at different times. The rows that show up in one query that are no longer present the next time the same query is performed are termed phantoms. Pessimistic locking of all rows returned in a result set for the duration of a transaction is required to prevent phantoms while non-repeatable reads can be prevented by locking a single row. Interestingly, locking of result sets to prevent phantoms, rows that disappear from a result set while work is in progress, does nothing to prevent the opposite problem, also referred to by some people as phantoms, where new rows appear that satisfy the query condition for inclusion in the result set while work is in progress. Phantoms of both types are also an issue problematic at times for indexing and traversing indices.

Durability is the property of a transaction and its various parts that guarantees changes made by the transaction will persist after completion and will not disappear unless the entire transaction is rolled back even if hardware failures occur and data is lost. Achieving near-perfect durability is difficult and expensive but not impossible. While nothing will save data from destruction when the next meteorite impact wipes out nearly all life on earth, the tools and techniques required to automatically replicate data off-site in real-time within a transactional context are well-known and widely deployed in situations where the definition of durability must include survivability of transaction data in the event of disaster that completely destroys a computing facility. Changes to information are made durable with the help of reliable transaction logging accomplished using force-write operations that have no chance of being cached in temporary volatile storage such as RAM but

instead are always written, by force, to persistent non-volatile durable storage such as a hard disk.

Windows On-Line Transaction Processing

In practice, On-Line Transaction Processing (OLTP) provides increased reliability only because the systems that implement it are designed to be robust both in terms of hardware and software. The only special procedural improvement that distinguishes most transaction-aware software from transactionless software is the optional ability to vote explicitly on the outcome of a transaction. Various techniques exist to enable each participant in a transaction to cast a vote on its outcome including several one-phase commit (1PC) protocols where each participant votes right away after being contacted by a coordinator and the votes are tallied, sometimes within a short timeout window where each participant either votes to commit within the timeout period or the transaction aborts.

Two-phase commit (2PC) is the most common transaction commit/abort decision-making protocol and 2PC is the transaction type implied generally by the use in present-day transactional computing of the term OLTP. In 2PC, a transaction monitor (TM) orders each transaction participant to enter a prepare phase where work to be performed by each participant is prepared by the participant but not yet committed to durable storage. Changes made during the transaction's prepare phase are hidden from other systems that are sensitive to the transitive nature of these pending changes because the second phase, the commit phase, may result in the transaction being aborted and all of the pending changes discarded.

Automated Transactions

Windows server operating systems provide native support for automatic transactions where transactional boundaries are automatically wrapped around transactional components so that the components need to have little or no knowledge of the implementation details of transaction management. Participants in automated transactions often give little or no regard to the transactional nature of the work in which they are engaged. An error or exception raised by a component results in an implicit abort vote during the commit phase, and the lack of an error condition implies a commit vote. The component may or may not have the ability to cancel or undo its pending changes, since the underlying resource against which the component does work may itself manage pending change rollback in the event of an abort decision for the transaction as a whole. Though only the software with final authority over persistent data storage may know of the existence of distinct transactions, the software components themselves are said to be transactional because the work they do occurs within a specific transaction context and is monitored by a transaction-aware OLTP-compliant resource manager (RM). The Windows service that provides automated 2PC transaction support is known as Microsoft Distributed Transaction Coordinator (DTC) and it was originally introduced for Windows NT 4 as part of Microsoft Transaction Server (MTS) supplied as part of Option Pack. In Windows 2000 and later, DTC from the optional MTS became the native COM+ DTC service.

Automatic transactions in COM+ (or MTS) are insecure because they rely solely on the exchange of GUIDs for authentication. Each transaction monitor (TM) is assigned a GUID

as is each resource manager (RM) at installation time or the first time they are used in a transaction after being installed. Each transaction is assigned a GUID dynamically. Together, these three GUIDs may seem difficult for an attacker to spontaneously discover or attack through brute force, but the algorithm used to create GUIDs isn't truly random (much of it is not even pseudorandom, incorporating discoverable information such as network adapter MAC address to ensure GUID uniqueness) and even if GUIDs were truly random, the DTC and each RM store their respective GUIDs unencrypted in the Registry or another easily-accessed location. Worst of all, these GUIDs are transmitted in the clear as transaction participants carry out their respective functions and exchange transaction state information or instructions with each other unless IPsec is deployed in the network. Even when IPsec is used, remote procedure calls that cross process boundaries but not network node boundaries often remain unencrypted and unauthenticated.

Distributed Transaction Coordinator Lockdown

The participants in conventional distributed transactions, where multiple transactional resources are managed by the COM+ DTC, trust each other implicitly. They have no automatic mechanism through which to determine who the other participants are because the DTC doesn't bother to provide this information and most transactional applications don't consider it a priority to deploy peer-to-peer communications between transactional participants. Identity authentication of transaction participants is typically left up to network security and the security context of the thread that contains the controlling root transaction context provides the only security principal against which each transaction participant can evaluate access permissions. Clearly this won't suffice in the event that network security is compromised because the controlling security context of the root transaction context is likely to be under the influence of malicious code. Attacks that succeed in executing bad transactions concurrently with valid ones are more severe than DoS attacks by many orders of magnitude since recovery requires time consuming analysis of transaction logs to manually distinguish between good and bad entries made at each transactional resource during the attack. To properly defend against this type of threat means automating the differentiation between valid and invalid transactions and cleansing transactional resources explicitly through execution of decontaminating transactions. Preventing contamination of transactional resources in the first place is a noble goal, but not one that can be accomplished realistically with absolute certainty, even when all transactional resources are protected by security-aware application servers.

The vulnerabilities that exist in DTC communication and its commit phase voting protocol due to the absence of encryption and authentication and reliance on keeping GUIDs secret are especially ironic considering the lengths to which the developers of the COM+ DTC went to make everything in COM+ securable. Every COM+ interface, every class and every object on every network node and within every process can be assigned separate activation and call security settings. As securable objects, everything in COM+ conforms to the Windows platform security fundamentals of SIDs, ACLs, DACLs, and ACEs. Permissions can be assigned based on role and group as well as to individual user security principals. In spite of all this, DTC itself is vulnerable to attack and spoofing when it comes to its own management of its trust relationships and its communication with other TMs and RMs. Windows XP and the .NET Server family make it possible, for the

first time since Windows' native OLTP facilities were created, to turn off all vulnerable exposure points for the DTC. The only communication path that remains is the one mediated by COM+ interfaces and local interprocess method invocation where both sides of any interaction have local security contexts and call stacks that can be analyzed automatically to enforce apparent security policies.

Figure 9-1 shows the new MSDTC configuration settings for .NET Server. To access this window, select Properties for My Computer inside the Component Services administrative tool. With the configuration window shown in Figure 9-1 you can set the default transaction coordinator to a remote host. If you do so, you'll want to implement IPSEC to secure the network used to communicate with that host, otherwise transaction GUIDs will pass to the remote transaction coordinator in the clear without authentication. Logging and tracing for DTC can also be configured. But the most important new settings available for the MSDTC are accessed by clicking on the Security Configuration button.

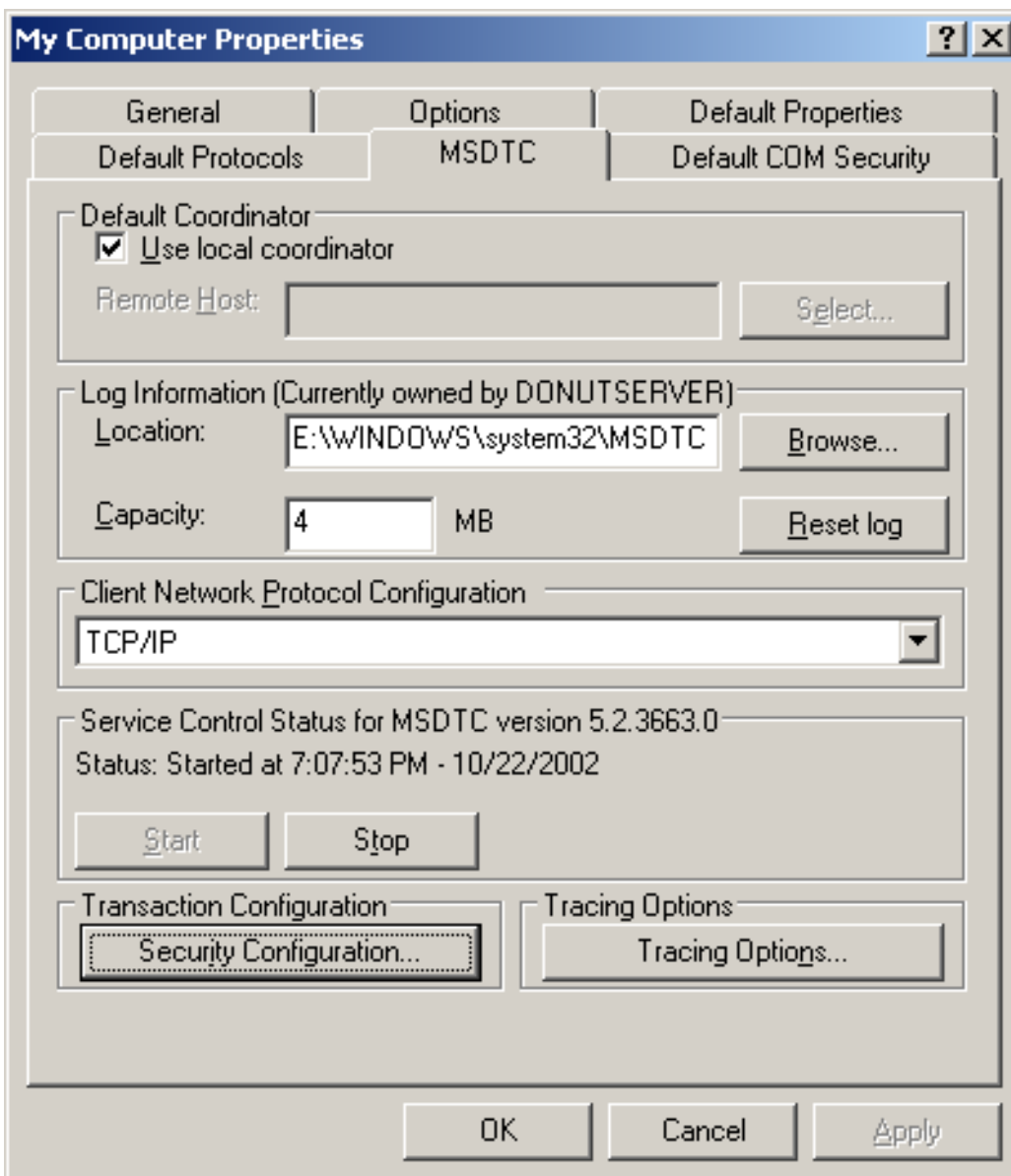


Figure 9-1: New .NET Server Component Services MSDTC Settings

As you can see in Figure 9-2, the new Security Configuration window for MSDTC in .NET Server allows you to completely disable the improperly-designed and insecure legacy DTC features. In addition, all network access and remote management capabilities are now disabled by default rather than enabled by default as in previous versions of Windows Server. XA Transactions are enabled by default, and you should disable them as well unless they're needed in your deployment to communicate with the transaction coordinator integrated with an RDBMS. The Transaction Internet Protocol (TIP) is also disabled by default, preventing certain COM+ Bring Your Own Transaction (BYOT) features.

When TIP is enabled in the MSDTC, the COM+ Bring Your Own Transaction classes can be used by malicious code to attack the DTC, forcefully abort transactions whose GUIDs are known or guessed, and produce DoS conditions. In particular, the COM+ interface ICreateWithTIPTransaction allows code to connect to port 3372 and join an existing transaction explicitly. The System.EnterpriseServices namespace in the .NET Framework Class Library includes a BYOT class that wraps this and other COM+ BYOT interfaces and class methods. Aside from the obvious DoS implications, the possibility that the DTC can be reprogrammed remotely to lie about the success of transactions if buffer overflow vulnerabilities are found make it critically-important to shut down all unauthenticated remote communication points.

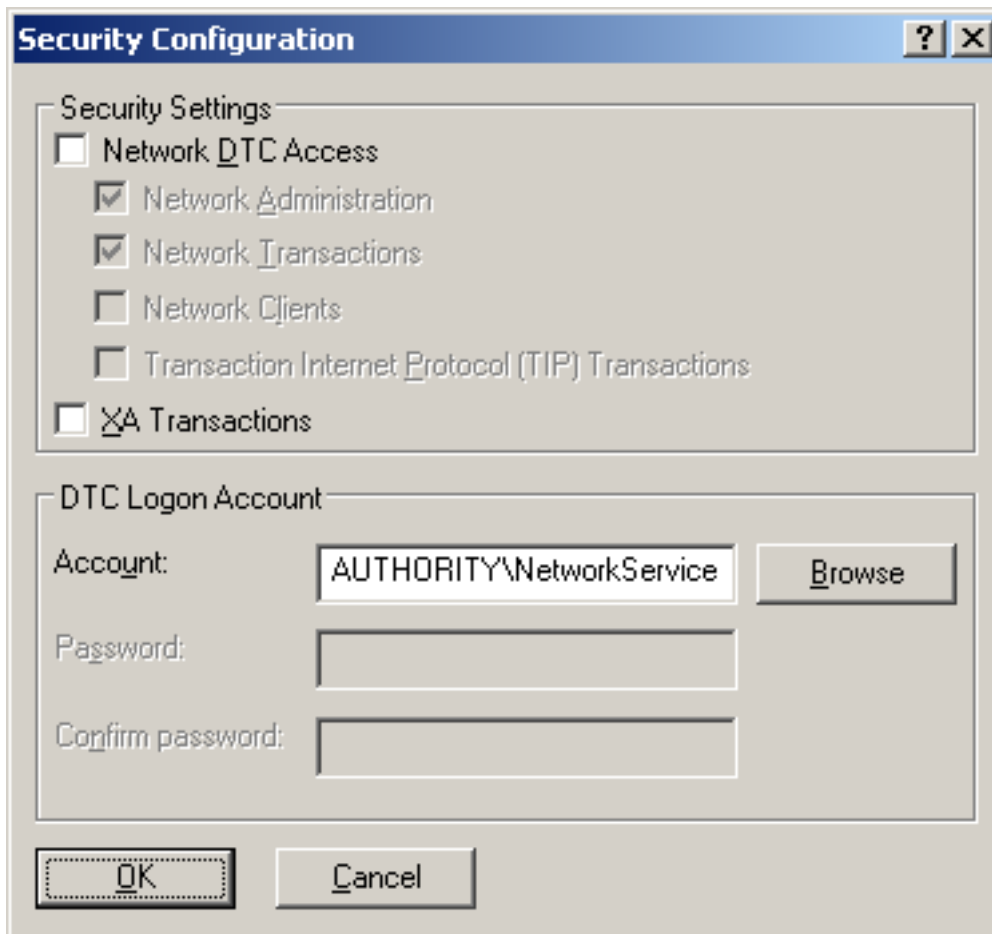


Figure 9-2: Disable Network DTC Access, TIP, and XA Transactions

To minimize the damage that can be done by a buffer overflow vulnerability, DTC no longer runs by default as LocalSystem but instead the unprivileged NetworkService account is used. Importantly, the NetworkService account by default has read-only access to DTC registry keys, making it impossible for a buffer overflow exploit that does not achieve privilege elevation to reconfigure the DTC. In Windows Server OS versions prior to the .NET Server Family the MSDTC was plagued with inadequate security and there was nothing you could do about it except cross your fingers and hope for the best or avoid the MSDTC entirely. IPSEC encryption deployed for your LAN was the best defense available, and it did little good when one of the hosts in the IPSEC-secured network became compromised because then nothing would stop it from attacking the DTC. With .NET Server these problems with the DTC have been resolved by default but the legacy functionality can still be enabled if necessary. For transaction processing within applications hosted by IIS, there are better ways to achieve distributed transaction processing security such as transactional message queues that benefit from the ACID properties required of transactions but avoid the architectural problems of combining Web-based applications with conventional OLTP services.

Crash Recovery of Pending Transactions

Interesting alternatives to the 2PC protocol exist that emphasize different aspects of performance and recoverability. One alternative worthy of note is called the Coordinator Log one-phase commit (1PC). In this protocol, every bit of work done by each transaction participant is communicated in detail to the transaction Coordinator where it is logged. In the event of a failure that impacts a transaction participant, all the participant needs to do in order to recover from the failure and resume its transaction processing duties is ask the Coordinator to replay any transactions it might have lost during the failure including any that may still be pending. The Coordinator Log 1PC protocol doesn't waste time with a prepare phase and then a separate voting (commit) phase, which reduces round trip overhead to and from each transaction participant. The drawback is also its strength: potentially large amounts of data representing every bit of work each transaction participant does is constantly flowing into the Coordinator Log just in case it's ever needed.

A common example used to explain transaction processing is the operation of a bank's Automated Teller Machine (ATM) network. Through transaction processing techniques it is commonly believed the bank account that a customer withdraws money from will have its balance reduced by the amount of the withdrawal unless the withdrawal fails or is cancelled. If the withdrawal occurs without the corresponding account balance reduction, transactional integrity is compromised and a bank error occurs in favor of the customer who receives money from the ATM without a corresponding reduction in bank account balance. OLTP makes an ATM network reliable, not foolproof. Several points of vulnerability remain where a sufficiently determined attacker, even one without special technical knowledge or skill, can still cause an ATM to dispense money without deducting the withdrawal from the affected bank account balance by creating a specific type of fault at just the right time.

OLTP doesn't stop financial crimes like theft, fraud, embezzlement, and forged financial instruments or stolen identities. It does enforce the fault tolerance rules of a particular

transactional architecture. In order to recover from a fault, each transactional resource must be able to read its transaction journal so that partially-completed pending transactions that halted due to a fault can be rolled back or committed properly to durable storage before allowing new transactions to occur. The only security this affords is a protection against data corruption caused by failures. It doesn't protect against pending transaction data loss nor does it prevent loss of data from previous transactions when a failure results in unrecoverable damage to both a database storage device and its transaction log. Nor does OLTP prevent database tampering or help to detect tampering when it occurs.

ATM machines initiate transactions by contacting a transactional resource that includes at least one transaction manager (TM) and one resource manager (RM) in addition to its own RM and possibly its own TM. If the TMs wait to commit a withdrawal transaction until after the ATM's RM confirms that cash was dispensed, an attacker may have a short window of opportunity to DoS the remote TM, withdraw cash, and reboot the ATM. This could cause the remote TM to rollback the pending transaction because it would never receive confirmation of the cash withdrawal from the ATM's RM or TM. Even if the ATM doesn't forget about the pending transaction as a result of rebooting, the remote TM may already have recorded an abort outcome for the transaction which leaves the ATM out of sync with the remote resource where definitive transaction record keeping is done.

This transaction model is known as presumed-abort. A presumed-commit model would be more appropriate in this scenario because a failure of the ATM to dispense cash could be verified by a bank manager in a variety of ways and this failure condition can thus be resolved without risk through human intervention. Upon deducting the withdrawal amount from the customer's account balance, the TM would commit the OLTP transaction and then instruct the ATM's RM to dispense cash. If the ATM never receives or can't process the dispense cash instruction the business transaction isn't complete but the OLTP transaction is, resulting in inconvenience (a customer complaint requiring human intervention to verify the failure and resolve) not theft. It is therefore more secure.

Hardening Business Processes

Threats to information security typically imply bad software and bad instructions to software that, if executed, would cause undesirable results. To build security into transaction processing the concept of what constitutes a threat must be let out of the small, optimistic box that defines infosec only in terms of computing as though other computers or other programmers are the only attackers that exist. The real world is full of things that explode, burn, break, melt, disappear out the back door, walk out the front door, sink irretrievably to the bottom of the ocean, and malfunction permanently when exposed to gunfire. These disasters can be more than just random natural or man-made catastrophes, they can also be tools of an attack on information systems. Reliable recovery from them without data loss or contamination is the main infosec role in transaction processing.

Designing and deploying secure transactional systems around IIS that can withstand any conceivable disruption, including the worst-case disaster recovery scenario of complete facility destruction, requires a slightly different perspective on transaction management than is typical. Most transaction processing systems presume that eventual recovery is possible. To facilitate recovery, transaction journals (logs) written previously to durable

storage through force-write operations, as opposed to lazy-write operations that could delay the actual write until some time in the future when a cache is flushed, are used to perform recovery operations. However, the infosec worst-case scenario is not just complete facility destruction, it is a successfully-executed security breach followed by the insertion of malicious transactions, extraction of sensitive data such as encryption keys followed by physical destruction of all evidence of the attack in a surgical strike that leaves carefully-concealed contaminated data and code to persist the effects of the security breach and enable subsequent rounds of attack.

Consider how a bank would deal with the unlikely scenario where a currency counterfeiter deposits \$20,000.00 in fake currency at an ATM and then blows up the ATM, destroying the counterfeit money but not the bank's deposit transaction record committed to an RDBMS by its transaction monitor. The attacker will demand that the bank honor the \$20,000.00 deposit, so bankers (and law enforcement) will review the video record as well as computer records of the transaction to determine the deposit's authenticity. Because the counterfeiter appears on the video record to be depositing authentic currency, and because the deposit record sent by the ATM to the RDBMS does not include sufficient information to prove the currency to be counterfeit, the bank may be obligated to honor the deposit and the attacker makes a profit if the explosives cost less than the amount of the fraudulent deposit. Transaction processing works perfectly in this scenario but doesn't save the bank or its insurer any money. Limiting the maximum deposit amount allowed per day per customer by way of cash deposits at the ATM to less than the market price of explosives sufficient to destroy the ATM would deter such a hypothetical attack by making it unprofitable.

Applications hosted by IIS have built-in access to conventional 2PC OLTP services. Although useful by itself for ensuring fault tolerance and data integrity, transaction processing must be supplemented by secure application design and business process security engineering to contribute substantially to information security. Before deploying any information system built around IIS that must preserve transactional integrity while processing HTTP requests you must first analyze the inherent security of the business process being automated. Even if you can't automate the detection of bad transactions or change a business process to make it easier to automate securely, you can divide any business process into individual OLTP transactions that are processed by different equipment with different security policies, passwords, and protections. The most common design mistake found in transactional Web applications is a desire by programmers to do everything that needs to be done to process a business transaction all at once in one place; usually at the point of first contact with the HTTP client. To be properly hardened against all possible threats while providing recoverability when attacks that are impossible to prevent occur anyway, Web applications must be deliberately break apart request processing into small steps even if it would be simpler and quicker to do everything all at once.

Preserving Common Sense Protections

It can't be emphasized enough in these pages that transactions are not security. Just because an application is transactional does not mean that it can be used safely by a wider range of end-users than a comparable non-transactional application. A transactional application is also no more trustworthy by virtue of its support for

transactions than it would be without them. It's common to describe software in terms of a producer-consumer relationship when the boundaries between client and server blur. One process produces data and stores it so that it can be accessed by a consumer process. Anything that can impact a producer poses a risk to all consumers, whereas risk to a producer is impossible as a result of a compromised consumer. It is harmful to security for producers to also be real-time consumers although they can safely be batch-mode consumers of filtered, verifiably-safe input. Anything a producer consumes during production can poison a product. When a producer must also be a real-time consumer, as in the case of an OLTP system that must accept live updates and make those updates visible immediately to other consumers, extreme caution must be exercised with respect to the transactions that the producer consumes.

The typical ATM withdrawal is an anonymous transaction in spite of the presence of a physical card and a PIN that together authenticate the cardholder's permission to withdraw cash from a particular bank account. ATM cards can be forged with relative ease, and a variety of techniques are used by attackers in the real world to capture PINs and reproduce an ATM card's magnetic stripe. Shoulder surfing is a common practice where a PIN number is captured as it is entered on the keypad through simple observation over the cardholder's shoulder. Skimming is the practice, used mostly by gas station attendants it seems, of capturing the contents of a card's magnetic stripe as it is swiped through a card reader when the authentic cardholder makes a purchase. ATM cards and credit cards that have been skimmed are easily cloned by the bad guys. Regardless of the attack method, the end result is that possession of an ATM card and knowledge of its associated PIN does not ensure that a person making a withdrawal is in fact the authentic account owner. This is analogous to the complex problems of authentication in any automated system including IIS. Every withdrawal is essentially made by an anonymous person, and every bank account that can be accessed through the use of an ATM card is at risk whether or not the cardholder ever uses their card. Transaction processing is critical to the operation of any ATM network but it doesn't protect anyone from bank fraud.

Bank accounts linked to ATM cards receive a measure of protection from the common sense security restrictions imposed by the bank. A daily limit on cash withdrawals prevents a thief from stealing large amounts of money from an account before the account holder is likely to notice a discrepancy and report it to the bank. Prompt and timely posting of transaction details to automated account information systems such as telephone-based interactive voice response and online banking services enable attentive account holders to identify unauthorized transactions within hours after they occur. This doesn't mean a bank account balance can't be tampered with through other means, but the ATM network is secure because it exposes only a finite transactional interface through which a limited number of commonplace banking activities involving small amounts of money can be accomplished automatically. Automated countermeasures to detect and stop ATM fraud are possible, such as disallowing a withdrawal attempt in Australia on the same day that another transaction occurs in the United States. Since a single cardholder can't be in both countries on the same day due to travel time and time zone differences, banks don't need any further proof of the existence of a forged ATM card and they can safely prevent further transaction activity in such a scenario. The same is true of ATM transactions that occur very close together in time at ATM locations in the same city that are separated by substantial distance. The authentic account owner could not possibly have traveled the

distance between the two ATMs in such a short amount of time, therefore one of the two transactions is fraudulent. An interesting and simple precaution taken by banks is automatic flagging for further investigation of accounts that exhibit context-sensitive transaction behavior that is known to be suspicious such as two back-to-back gasoline purchases at different gas stations. Banks use common sense to know that you can't fill your gas tank twice. IIS should impose similar common sense security restrictions when hosting transaction-aware applications.

Creating Transactionless Idempotent Operations

An idempotent operation is one that results in the same outcome no matter how many times it occurs. An instruction to set a variable equal to a fixed value is idempotent but setting a variable equal to its existing value (whatever it happens to be) plus a fixed value is not idempotent because redoing the same operation multiple times will result in different outcomes. Nearly every automated process can be broken into distinct idempotent operations, and there are good reasons to consider this design approach compared to conventional transactional design when it comes to transaction processing in Web applications. When a Web application is driven by end user requests it is subject to the pitfalls of the browser refresh button, user tendency to double click hyperlinks or buttons when they should only single click them, and the ability for the user to move back to any previous step and redo arbitrary pieces of the data entry process. Experienced users will even know how to open the target of a hyperlink in a different window and thus branch a logical path into two concurrent parallel paths through the application's request processing logic.

A core design decision that must be made by Web developers is whether to grant the user complete freedom to experiment with possible outcomes or whether to code application logic that catches every possible invalid or mutually-exclusive request sequence and forces the user to choose a single self-consistent path through the application at all times. When transactions are idempotent, the user either makes them happen or she doesn't and there is no need for further coding to determine whether or not a certain transaction is legal in the context of what the user is doing at a given time. Redoing the same idempotent transaction over and over again results in no harm to the integrity of the information system and requires no special coding to accommodate properly. This is a valuable characteristic for Web applications because it matches the usage model of browsing back and forth and refreshing and spawning new browser windows to explore tangents. Rather than building a system that breaks when data relationships aren't consistent at all times, idempotent operations lend themselves to allowing the user to push data around to where it makes sense to them and then requesting that some action be taken by the system with respect to the data.

The final step in an on-line shopping application would be a request by the end user for the merchant to bill them and fill the order as specified. To design this step as an idempotent operation your application would set a Boolean flag indicating that the customer has accepted the order and it is ready to be fulfilled rather than adding a new order record to a table. If the customer repeats the same process minutes later with different items, a single larger order could be the result rather than two separate orders. There's no need to force the customer to be done shopping and make them start over again when your application is designed around idempotent operations rather than conventional

transactions. The customer in this scenario doesn't think in terms of transactions, anyway, other than the single larger concept of shopping at a merchant's store. The unacceptable risks of exposing a credit card processing oracle for attackers to abuse have already been noted previously, so there can be nothing gained by forcing real-time credit card processing, either, unless there is an idempotent design here as well. Instead of approaching the credit card authorization with the idea that you are going to "bill the customer for their order" design it in terms of "setting the authorization level to the amount of the sale" instead. The former is transactional, while the latter is idempotent and benefits from the ease of implementation afforded by the fact that resetting the credit card authorization level for a transaction can be repeated over and over again without producing unnecessary transactional clutter. If the customer's credit card has already been authorized for an amount greater than the amount of the sale because of previous rounds of experimentation by the customer in the checkout process, the idempotent operation results in no change and it's not difficult to determine that this is the proper outcome of the customer's request. Compare this to the querying and accounting record analysis that has to take place in the transactional approach where previous orders may need to first be canceled in order to accept and process the updated sale.

Hardware Foundations of Secure OLTP

Microprocessors don't restrict certain locations in RAM such as stack frame memory from being used as a source of machine code instruction input. To compensate for this security oversight, some operating systems' memory manager routines disable execute permission for the stack memory segment of each process. Such OSes are said to provide a non-executable stack. As OS vendors deploy 64-bit processor support it is increasingly common for 64-bit binary executables to conform to the non-executable stack rule explicitly. An OS that provides a non-executable stack automatically extends stack buffer overflow protection to every software program that is executed including all dynamic libraries loaded by a process. A very small number of programs already in use today that are compiled to target pre-64-bit architectures are designed to dynamically store machine code instructions in stack memory on purpose and then redirect program execution so that the processor instruction pointer references these machine code instructions located in the stack memory segment. There are other ways for such software to accomplish dynamic machine code creation and execution, as through the use of dynamically-allocated heap memory, but there may be applications that just can't be ported from an executable stack to a non-executable stack for one reason or another. Even if non-executable stack permanently kills a few useful programs this is an acceptable loss to prevent an entire class of real-world software-only attacks that result from software development malpractice.

With the right knowledge and tools it's conceivable to throw a buffer overflow exploit at an ATM. What that would get you as an attacker is quite likely nothing but the ability to create a DoS condition that impacts other users of the ATM. Suppose it did give you the ability to forge authorized requests for money from arbitrary bank accounts. How would a bank's information security engineering team protect against such a threat when the ATM's microprocessor doesn't offer any help in hardware and meticulous review of software and firmware source code may not remove all such vulnerabilities in the system? The answers are plentiful and most require no change to the ATM (client) because they can be implemented entirely on the server side. For starters, variable rate

limits can be set based on a standard deviation of the moving average of transactions per minute received historically from each ATM during particular days of the week and particular months of the year. Statistical aberrations in transaction volume, ratio of withdrawal transactions to all other possible transactions, and repetitive withdrawal amounts can trigger lock down of a compromised ATM after as few as two transactions. The negative impact of a false positive lockdown trigger is limited to inconvenience for an innocent customer, while the protection against substantial theft is near-perfect.

To apply this thinking to your IIS deployment, you have to realize first of all that the belief that applications hosted by IIS must have the ability to survive, and service, a massive onslaught of authentic transaction volume (the Slashdot effect) is usually unfounded. A number of transactional Web applications truly need this ability, such as on-line brokerages that must be able to process transactions during panic buying and panic selling. However, these businesses always have backup failsafe confirmation processes and time periods after such transaction spikes where suspicious transactions, or even a suspicious cause for all transactions, can be pinpointed and if necessary certain transactions reversed. A business process that is not hardened thus would be inappropriate to automate and does not need the ability to continue processing transactions when volume exceeds the point where a business can no longer apply its failsafe protective processes due to time constraints. Refusing to accept for processing any transaction that will not benefit from failsafe business mechanisms is an absolute necessity for every secure transactional Web application.

Write-Ahead Logging and Shadow Blocks

Hard disks implement transactions too. The loss of power during hard disk write operations is impossible to avoid, so the hard disk must be managed by its device drivers in such a way that transactional atomicity is guaranteed. Consistency and isolation are also necessary because a variety of concurrent processes will always read and write asynchronously, creating the potential for overlapping or conflicting changes that can also result in dirty writes, dirty reads, non-repeatable reads, and phantoms. Durability is obviously a requirement of any hard disk transaction, we can't have hard disks forgetting about data written to them or spontaneously remembering data that was supposed to have been removed from the active filesystem. Several different techniques have been developed to achieve ACID transactional quality for hard disk operation and each depends on the durable property of the hard disk plus the smallest unit of atomic operation possible for writing to the disk, the block. A hard disk is designed so that it can never pause the writing of a block to service an interrupt that requires it to do something else before completing the writing of the block. The disk could lose power before the block is completely written, or it could encounter a physical defect while writing the block, but like a microprocessor executing a single machine code instruction, a hard disk executing a single block operation is a truly atomic operation with respect to that hard disk.

A common technique for hard disk transaction processing known as write-ahead logging implements a 2PC protocol with a coordinator log feature. A begin record is written to an append-only log stored in a special location on the hard disk followed by complete redo and undo information for the changes to each block modified by the transaction. The changes to each block are actually attempted only after their corresponding log entries are written to disk. If a crash occurs before a commit record is written to the log, a

recovery process can verify that transactions previously committed to the disk are still present and any partially-completed transaction can be rolled back. The processing of a recovery log is an idempotent operation; repeating the recovery process multiple times results in the same outcome. Idempotency of the recovery process is critical because additional failures could occur during recovery that require recovery to restart after the new fault is resolved.

Another common technique for hard disk transactions is known as shadow blocks. Instead of making changes to existing blocks, every write operation results in new blocks. Each block is a node in a durable linked list. The blocks each reference the next block in the list which makes it possible for changes to be made anywhere in the filesystem by writing a series of shadow blocks which link to themselves internally but where the final block links back to an original block that did not need to be changed. By linking to the first shadow block from the block just prior to it in the durable linked list, which requires replacing only a single existing block and is therefore a truly atomic operation, the modifications stored in the shadow blocks become a part of the active filesystem and the old blocks become free space. If a fault occurs before the shadow blocks are linked in, the shadow blocks are marked as free space again during recovery and the pending changes are lost because the commit (overwriting the original block just prior to the first shadow block) did not occur before the fault. If a fault occurs just after the linking block is overwritten, the recovery process views the original blocks as free space instead. In either case, the ACID properties of both the truly atomic operation where a single block is changed in order to link in the shadow blocks and the entire transactional change to the filesystem represented by the shadow blocks are guaranteed.

Transaction Logging to Facilitate Decontamination

The only way for a system that processes transactions to be completely secure is for it to presume that it will eventually process malicious transactions and that when this happens there will be only one way for these transactions to be undone: manually, through human intervention. It must be the goal of an automated transactional system, then, to log as much information as possible about the transactions it processes. Logs generated by transaction processing need not be reviewable for security breaches by automated systems, and they may not even be read, ever, by any human. Such detailed transaction logs exist to ensure that there is sufficient forensic evidence of what happened and why during an incident response that examines specific time periods or specific transactions. Some of this log information needs to be kept on-line, along with the data itself, to facilitate more common types of decontamination that may be required such as undoing all changes made by a certain user account during a certain time period. But most security log data can be sent off to a logging server that merely archives logs and doesn't make them available as additional fields or related records within the live data processing system.

Replicating Logs with Point-to-Multipoint Communications

Intruders know how to tamper with or erase conventional logs like the Windows Event logs. They also know how important it is for them to do so in order to avoid detection. With transaction processing it's even more important to protect logs from tampering because they are often the only records kept as a transactional resource is changed by completed

transactions. Fine-grained recovery and rollback operations are impossible without transaction logs. Because security is in the eye of the transaction coordinator, and the coordinator only does what it is told, transaction journals kept by transaction participants in order to facilitate failure recovery should not be mistaken for secure transaction logs. Transaction logs contain details about each transaction but lack application-specific security context, particularly in the case of failures. Additional logging is always necessary to ensure security, and it's never a good idea to write logs only to one place or store them locally because doing so leaves security logs vulnerable to tampering when the system is compromised by an intruder.

Multipoint communications through multicasting makes it possible to transmit log entries over the network to an indeterminable number of hidden nodes. Unlike relying on a single logging server that is easily discovered by an intruder, logging through multicasting makes it possible to conceal the existence and location of systems that capture and log data. This makes it nearly impossible for an outsider who does not possess knowledge beforehand of the whereabouts of and vulnerabilities in or access points to logging servers to even begin attempting to tamper with logs. The following code shows the use of .NET Framework classes to bind a `UdpClient` object to a multicast group address, 225.1.1.1, in order to receive data that is sent by any number of multicast senders. The `UdpClient` class includes a `JoinMulticastGroup` method that sends out an Internet Group Management Protocol (IGMP) membership announcement to alert the local multicast router, if any, that an endpoint on the host now belongs to the multicast group identified by the IP address specified in the IGMP announcement. Most important, the local TCP/IP stack associates the local endpoint with membership in the specified multicast group and will henceforth deliver all UDP packets addressed to the multicast group address to `UdpClients` (or sockets) that have requested membership in the multicast group.

```
byte[] c;  
IPAddress mcastaddr = IPAddress.Parse("225.1.1.1");  
IPEndPoint ipep;  
UdpClient mcast;  
ipep = new IPEndPoint(IPAddress.Any,2525);  
mcast = new UdpClient(ipep);  
mcast.JoinMulticastGroup(mcastaddr,2);  
c = mcast.Receive(ref ipep);  
System.Console.Out.WriteLine(  
    Encoding.ASCII.GetChars(c,0,c.Length));  
mcast.Close();
```

The second parameter to `JoinMulticastGroup` is a time-to-live value. By default multicast packets have a TTL of 1, which prevents routing across multicast routers; only multicast group members on the same network segment receive such packets. Increasing the TTL beyond 1 allows additional multicast routers to replicate the traffic if the router can reach multicast group members eligible to receive the packets. The `Receive` method is used to block the calling thread until a datagram arrives addressed to one of the `UdpClient`'s multicast addresses. The code as shown simply displays the datagram message received in the packet using the console's `WriteLine` method. After the `UdpClient` has joined a multicast group, any endpoint that has the ability to employ multicast routing can deliver datagram packets to the `UdpClient`. Multicast senders do not need to first join the

multicast group to which they transmit datagrams. The following code transmits a datagram message to the multicast group identified by address 225.1.1.1. The message is routed to each member of the multicast group that is located on the same network segment. To route the message beyond the local network segment the `UdpClient` must first specify a TTL for its message either by explicitly joining the multicast group it's sending messages to or by setting the `SocketOptionName.MulticastTimeToLive` socket option on its encapsulated `Socket` with the `Socket.SetSocketOption` method.

```
byte[] b;  
IPAddress mcastaddr = IPAddress.Parse("225.1.1.1");  
IPEndPoint ipep;  
UdpClient mcast;  
ipep = new IPEndPoint(mcastaddr,2525);  
mcast = new UdpClient();  
b = Encoding.ASCII.GetBytes("message");  
mcast.Send(b,b.Length,ipep);
```

Port number 2525 and the multicast IP address shown here were selected arbitrarily. Like any IP communication, port number determines specific application endpoints at particular IP addresses, and only a `UdpClient` (or UDP socket) bound to the same port number and multicast group (IP address) as the sender's multicast UDP packet will receive the multicast message. Multicast packets are more easily routed to an arbitrary number of logging hosts than are broadcast packets. The IP multicast class D address range for group identification (224.0.0.0 to 239.255.255.255) makes for simpler multipoint application development because packets addressed to the group IP address don't have to be multiplexed between a variety of different multipoint-capable applications on each node that may all share the use of the single broadcast address. Of the class D address range reserved for IP multicast, the entire class C address block 224.0.0.0 to 224.0.0.255 is reserved for control messages and other multicast management functions. A few others outside this class C block are well-known multicast group addresses reserved for use by certain applications such as clock synchronization. The Internet Assigned Numbers Authority (IANA) maintains a comprehensive list of current reserved multicast group IP addresses. Figure 9-3 shows a Network Monitor capture of the three multicast packets sent by the code shown here. The first packet is an IGMP group membership announcement that results from the call to `JoinMulticastGroup`. The second packet is the multicast message itself and the third packet is an IGMP Leave Group Message. The IGMP packets are used only by multicast routers if any are present on the network segment.

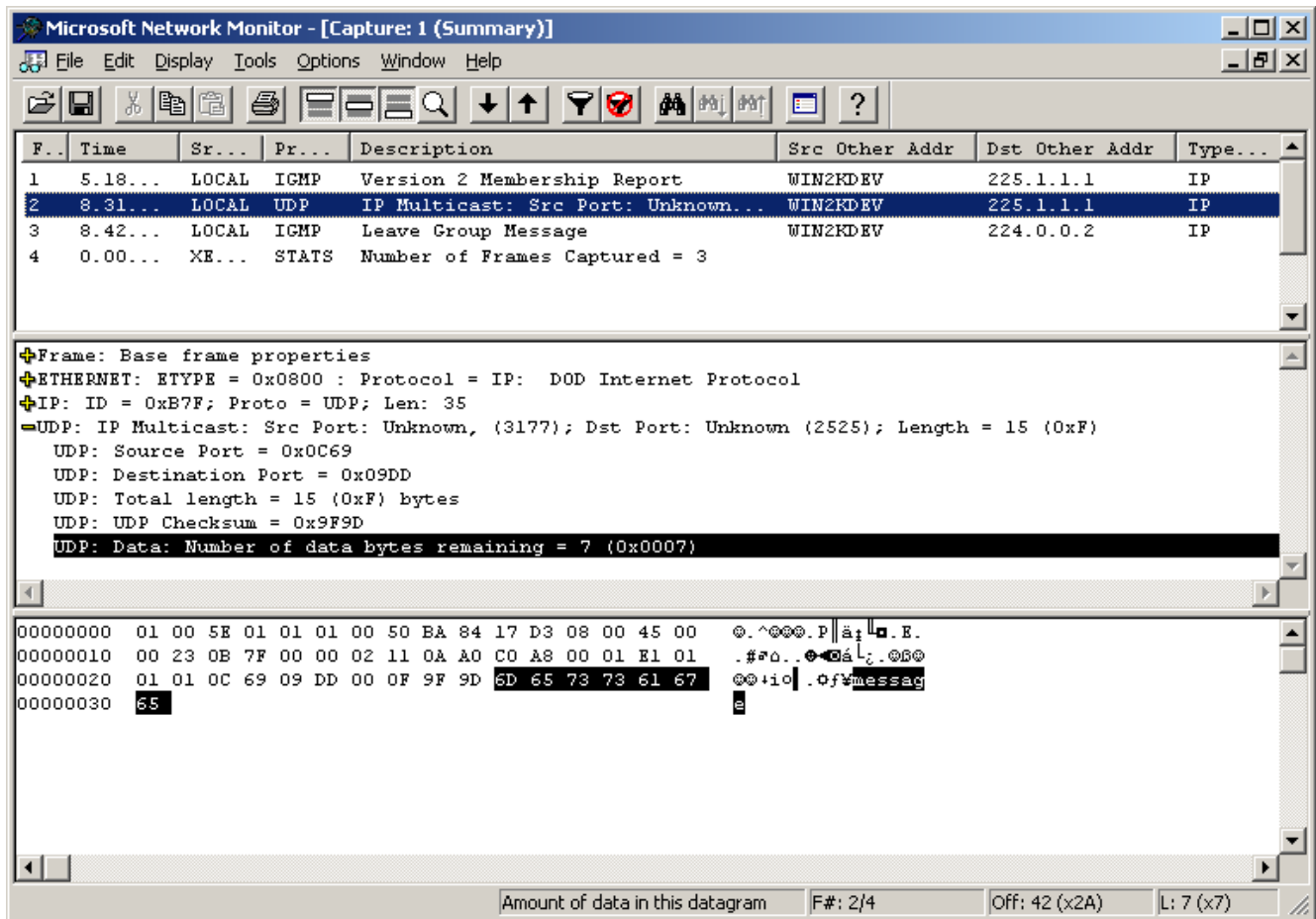


Figure 9-3: Multicast UDP Packets Are Surrounded by IGMP

IGMP may enable discovery of each member of a group within the local network segment managed by a particular router, but one-way inter-segment multicast routing is possible between networks in order to prevent an intruder from discovering specific nodes that operate as logging servers and are members of a multicast group. Multicast routers need not relay all IGMP messages they receive from local network nodes. In particular, IGMP membership announcements don't need to be forwarded to the entire multicast group, the multicast router simply needs to keep track of whether or not there are still multicast recipients participating in a particular group to decide whether or not to disable upstream multicast route delivery of the group's datagram traffic.

Tracking Responsible Security Principals

One of the most important data points to collect about every transaction is the apparent security principal that controls the transaction initiator and each of the transaction participants. Only through logging of this information can selective decontamination occur during incident response. Ideally you would ensure that different components responsible for certain types of changes to data execute under different user security contexts at the process level and then impersonate individual, less-privileged users at the thread level. Logging apparent security principals for both the process and thread provides valuable information about the source of security incidents and helps find and eliminate contamination.

It's important to log security principals on both sides of any instruction that results in durable changes to valuable data. A call to a database server made by a user security context that is suspicious on the client-side won't look suspicious if you only see the user account that the caller used to authenticate with the database server. On a database server you might define tables using a default constraint that populates a column with the user ID as perceived by the server. Something like the following is a common way to create such a default constraint for a table.

```
CREATE TABLE TRNLOG  
( trnlogid int IDENTITY(100, 1) NOT NULL,  
  userid varchar(255) NOT NULL DEFAULT USER )
```

Stored procedures and triggers are also a good way to force-log additional information about each transaction. Remember, though, that an intruder may succeed in tampering with anything contained in the databases controlled by a given server. Failsafe logging that originates from other points in a transaction including at the locale of every transaction participant is important as well. For especially sensitive information systems it isn't uncommon for real-time network traffic logging to occur with a specialized network analyzer. Such systems usually rely on a circular log device of fixed size so that only a snapshot of recent activity is captured. Otherwise the cost per transaction is increased far beyond the incremental improvement to security provided by complete logging of all network traffic due to the cost of very large high-performance storage devices.

Logging The Call Stack

One of the most valuable and least used security logging techniques is recording the call stack at various times during program execution. Capturing and logging call stacks on-the-fly gives you a way to analyze during security audits and intrusion incident response more detailed information about the code that executed at various times in the past. The easiest way to read the call stack is through the use of the .NET Framework class library namespace System.Diagnostics. The following code reads the call stack and writes it to a System.IO.MemoryStream buffer.

```
byte[] b;  
StackFrame sf;  
MemoryStream ms = new MemoryStream();  
StackTrace st = new StackTrace();  
for(int a = 0;a<st.FrameCount;a++) {  
  sf = st.GetFrame(a);  
  s = sf.ToString();  
  b = Encoding.ASCII.GetBytes(s);  
  ms.Write(b,0,b.Length); }  
}
```

The System.IO.MemoryStream class provides a WriteTo method that writes the entire contents of the MemoryStream to another stream. Using WriteTo you can easily dump the call stack contained in the MemoryStream to a file or network stream. With the ability to read and use the call stack on-the-fly at runtime comes new possibilities for preserving

all information necessary to pinpoint transactions that occur while under the influence of malicious code.

Hashing The Call Stack

A valuable security technique that comes from call stack logging is the ability to hash the call stack and send the resulting hash code to any service that carries out processing on behalf of the pending transaction. Each transaction participant can easily store a list of the authorized call stack hashes, one of which must be provided by the caller in order for the transaction participant to carry out the requested operation. The following code shows how to hash the call stack through the use of a MemoryStream object. When you set out to deploy call stack hashing, it's important to log the call stacks and resulting hashes produced through normal operation of your application. This is important in order to develop a profile of the call stack hash codes that are expected to occur during normal, uncontaminated, trusted execution flow. Every change to source code that impacts bytes in memory prior to the code that reads the current call stack will result in slight differences in memory offsets for each function call. Because the precise memory offsets of functions called prior to the one that reads the hash are critical forensic evidence of the integrity in memory of the code you expect to be executing, these minor changes result in very different hash code output when the call stack is hashed.

```
HashAlgorithm md5Hasher = MD5.Create();
byte[] hash, b;
StackFrame sf;
MemoryStream ms = new MemoryStream();
String s = Process.GetCurrentProcess().ProcessName;
b = Encoding.ASCII.GetBytes(s);
ms.Write(b,0,b.Length);
StackTrace st = new StackTrace();
for(int a = 0;a<st.FrameCount;a++) {
    sf = st.GetFrame(a);
    s = sf.ToString();
    b = Encoding.ASCII.GetBytes(s);
    ms.Write(b,0,b.Length); }
ms.Position = 0;
hash = md5Hasher.ComputeHash(ms);
System.Console.WriteLine(BitConverter.ToString(hash));
```

Hashing the call stack and using the resulting hash code as a form of authentication to authorize remote processing on behalf of the caller does not prevent malicious code from sending a forged call stack hash. It does provide an automated defense against code tampering and code injection attacks that alter the call stack and thus give themselves away. An attacker who intercepts authentication credentials won't be able to send requests using those credentials without also intercepting or discovering the right call stack hash code required to send a particular transaction request to a transaction participant.

Something similar can be accomplished by hard-coding a unique number, a shared secret, in both the transactional resource that carries out processing and the caller that requests it.

However, such shared secrets don't provide the extra protection against misappropriation of the compiled software through buffer overflow attacks and other techniques that inject foreign code into a legitimate process. The code shown previously hashes the contents of a MemoryStream buffer after writing the call stack to the buffer and appending the name of the current process. In addition, the Process class in the System.Diagnostics namespace includes a MachineName property you could add to the MemoryStream buffer prior to the ComputeHash method call in order to produce a different hash code depending upon the name assigned to the computer the process is running on.

The following source shows an even better malicious code defense for transaction participants that includes in the MemoryStream buffer the complete list of modules loaded into the process. In most cases, the list of modules loaded by a process is known at all times and any deviation from the known module list suggests the presence of malicious code. An example of a situation in which the module list is not static and therefore predictable is where an application process pool results in many applications sharing the same process, a pool of threads, or pool of processes. Even this scenario will not impact predictability of a module list, however, if the applications that share the pooled resources don't dynamically load any modules explicitly and all dynamic modules loaded implicitly are loaded when the applications all finish launching. In such a case the loading of any unexpected module should be considered fatal to the security integrity of the entire pool, and intentionally halting program execution based on this simple security policy violation trigger makes a great deal of sense to protect each application in the pool.

```
HashAlgorithm md5Hasher = MD5.Create();
byte[] hash, b;
StackFrame sf;
MemoryStream ms = new MemoryStream();
String s = Process.GetCurrentProcess().ProcessName;
b = Encoding.ASCII.GetBytes(s);
ms.Write(b,0,b.Length);
StackTrace st = new StackTrace();
for(int a = 0;a<st.FrameCount;a++) {
    sf = st.GetFrame(a);
    s = sf.ToString();
    b = Encoding.ASCII.GetBytes(s);
    ms.Write(b,0,b.Length); }
foreach(ProcessModule pm in
    Process.GetCurrentProcess().Modules) {
    s = pm.FileName;
    b = Encoding.ASCII.GetBytes(s);
    ms.Write(b,0,b.Length); }
ms.Position = 0;
hash = md5Hasher.ComputeHash(ms);
System.Console.WriteLine(BitConverter.ToString(hash));
```

By using the Modules property of the Process class the complete list of modules loaded into the active process is obtained and included in the MemoryStream buffer before computing its hash code. This list can only be tampered with at great difficulty such as through the installation of a rootkit Trojan. Hashing the module list in addition to the

process name and call stack and providing the resulting hash code to transaction participants gives them a way to know precisely which code is responsible for initiating each request. With this knowledge, an appropriate security policy and privilege restriction can be imposed and automated lockdown procedures can be triggered when a request violates these fixed policy rules. Strict limits on allowable requests for each security principal are always important, but in practice only a subset of all allowable privileges are used by a particular client process and the use of call stack hashing to differentiate between these different contexts can be valuable.

As noted previously, any change to source code that results in changes to compiled object code's size and relative position in memory is likely to change the call stack and thus its hashed value. This presents something of a catch-22 if you intend to deploy such a countermeasure because it can be difficult to discover what the hash code is, in order to use it for something, without dumping the hash code to a file or other output every time the program runs. The trick is to output the stack hashes one at a time by inserting lines of code after the point where the hash is computed, rebuilding the project, and running the program until the hash code is output through the logic you've temporarily inserted. Then relocate the hash output logic to just following the next hash computation and repeat. Be careful not to declare any local variables or do other things in your hash output logic that would impact static memory allocations for the function or you will modify the position in memory where the hash computation occurs relative to its parent function entry-point. The `System.Console.WriteLine` instruction shown in the sample code from this section works well for this purpose as it references only the existing variable named `hash`.

Using a Call Stack Hash as a Secret Key

Once you've computed the hash code of the current call stack, you can easily use the bits of this hash code as a secret key in symmetric encryption. A hash code of this sort is useful only for protecting a shared secret or a private key that must be stored online within automated reach of software. A random (or pseudorandom) shared secret key must still be generated and exchanged between parties who need to conduct encrypted communications or those parties must exchange their public keys and use asymmetric encryption as a bulk cipher as shown in Chapter 3. Each callee must be given the shared secret ahead of time in order to decrypt any ciphertext communication it receives from the caller, but instead of storing the shared secret in cleartext on the caller, which is typically an application hosted by IIS, the shared secret can be stored as ciphertext encrypted using the caller's call stack hash.

It's important to understand that this protection of the hard-coded shared secret, while better than nothing and interesting because of the tamper-resistance it provides for the compiled code compared to other techniques, uses a key that is not even pseudorandom. Though the key is very large in terms of its bit-length, the possible keys that will ever be selected are limited by the possible variations in call stacks, module lists, and process names from one application to the next. This possible key set is very small compared to the key's bit-length, and an attacker who is able to conduct a forensic analysis of call stacks could create a dictionary of possible call stack hashes that would be substantially smaller than the full range of possibilities represented by the key length in bits. In this way, the use of a call stack hash code as a symmetric encryption key is useful mostly

through security by obscurity as an automated defense against any code injection or other attack that gives itself away by modifying the call stack. Such attacks would disable the communication ability of the software being attacked by virtue of the fact that the call stack hash code is no longer equal to the secret key used to protect the shared secret and thus the code, while under attack, is unable to decrypt the shared secret or the private key it needs in order to communicate with other code.

Transactional Message Queues

You have teeth and chew your food because swallowing food whole would remove a crucial life-saving protective measure that enables your stomach to filter out sharp objects, caustic chemicals, and other harmful non-digestible materials. Many transactional Web applications take data that should be chewed (validated over time in successive steps) and use it immediately as though digestion isn't necessary and faster is somehow better. Web applications that function as consumers as well as producers without chewing their food will likely die and be removed from the gene pool. To keep transactional Web applications alive and healthy, use transactional message queues as barriers to halt real-time processing as transactions are validated. This separates the producer and the consumer in a transaction by space and time, and enables the consumer to carefully chew on a producer's output before deciding to swallow it. The use of transactional message queues also enables the preservation of transactional integrity for a business transaction without forcing every facet of a business transaction to occur in real-time as part of a single OLTP transaction.

A message is any data, including a serialized object instance, that is sent from a message sender to a message queue where a message receiver is able to retrieve the data at a later time. The delay between send and receive can be short or long and during this delay interval a queue provides a storage container for messages that have been sent but not yet received. Queues can be public or private, automatically logged in a journal or audited, they can require encryption, enforce access restrictions and perform authentication, and even require digital signatures for each message. Send and receive operations can even be made transactional so that a fault or processing exception defined arbitrarily by application requirements or business logic will result in no change to the queue or its pending messages. Transactional message queues can be incorporated into distributed transactions and serve as an important practical safety measure in the operation of real-world Web-based transaction processing systems.

Microsoft Message Queue Service

The System.Messaging namespace in the .NET Framework class library provides support for the Microsoft Message Queue (MSMQ) service. MSMQ supports transactional message queues as well as encrypted, digitally signed and authenticated messaging. Applications hosted by IIS can access MSMQ services through the .NET Framework as well as through COM+ interfaces. In addition, the MSMQ queues and COM+ interfaces to them are securable objects with distinct DACLs, SACLs, and ACEs. Queues can be created dynamically, managed remotely, and logged completely with journals. Delivery confirmation is possible through a special dead letter queue and transactional messages sent to remote queues can provide end-to-end message delivery tracking. Secure messaging with MSMQ is superior in some respects to direct method calls for the

purpose of kicking off processing due to its enhanced usage and logging scenarios as well as alternatives to conventional Windows authentication for creation and receipt of messages.

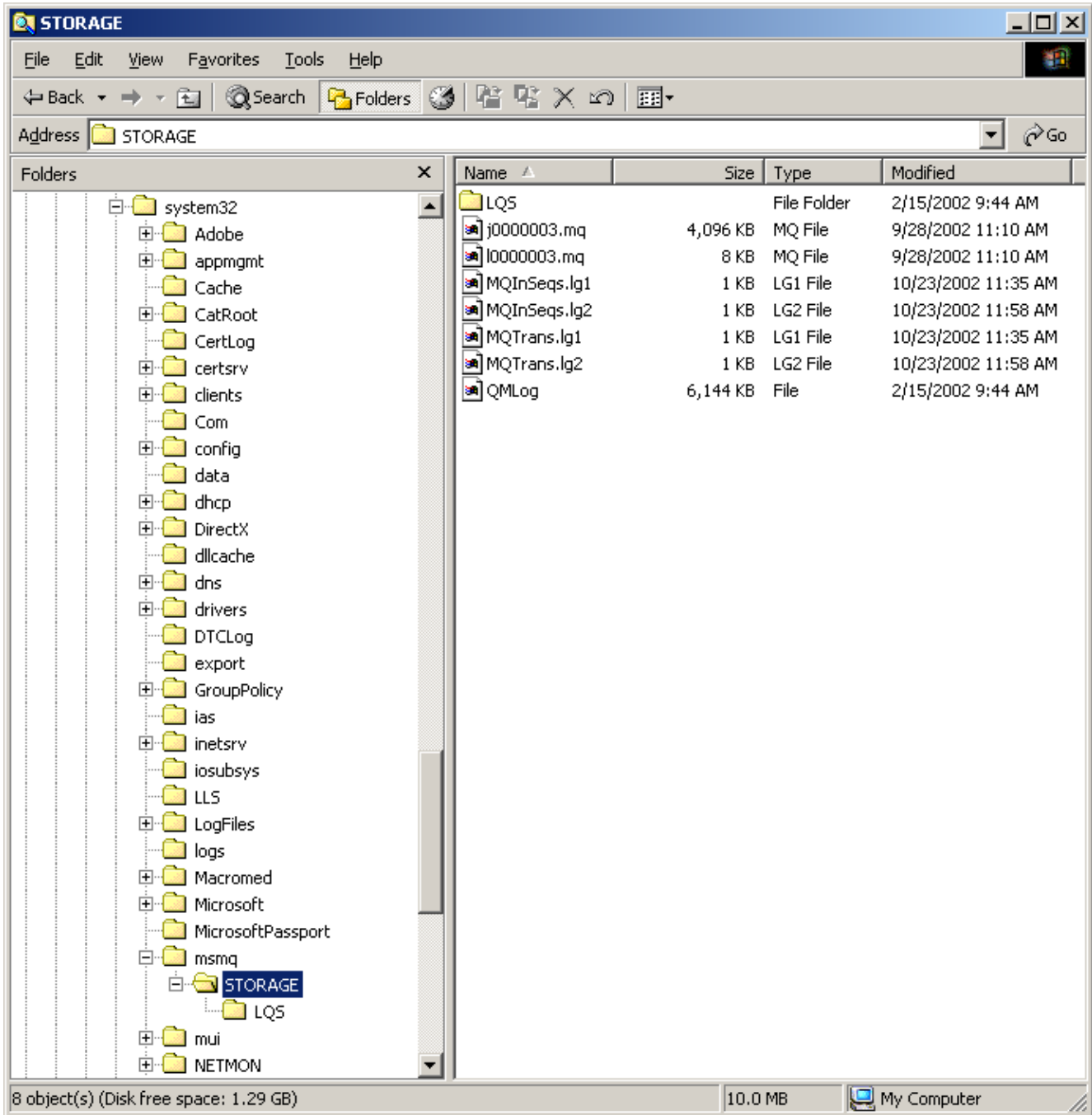


Figure 9-4: Filesystem Storage of Queued Messages

Messages stored in MSMQ queues are saved by default in the %System32%\msmq directory as shown in Figure 9-4. MSMQ queues are either user defined or system queues and each can be either public or private. Private queues never propagate messages between queue servers and the messages they contain are normally used only on the local box that contains the private queue. Public queues publish themselves publicly for use by applications locally or, if allowed, also from remote locations. Messages stored in both

private and public queues can be retrieved from the queues by authorized receivers provided that the receiver knows the full path to the queue. The following C# code creates or connects to a private unencrypted message queue without authentication and enables logging of messages in a journal.

```
MessageQueue mq;
if(!MessageQueue.Exists(".\\Private$\\queue3$")) {
mq = MessageQueue.Create(".\\Private$\\queue3$",true);
mq.UseJournalQueue = true;
mq.EncryptionRequired = EncryptionRequired.None;
mq.DenySharedReceive = false;
mq.Authenticate = false; }
else {
mq = new MessageQueue(".\\Private$\\queue3$"); }
```

The MessageQueue class belongs to the System.Messaging namespace in the .NET Framework class library. Its Create method adds a new message queue at the specified path with the specified queue name and marks it as transactional if the Boolean parameter is true. A transactional message queue is one that requires a transaction to be present when sending or receiving messages and will automatically wrap any non-transactional single message send or receive requests in a presume-commit internal transaction that does not give a chance for the non-transactional caller to abort the transaction if something goes wrong. Internal consistency and fault tolerance are thus preserved for the transactional queue but not for the non-transactional application that uses it. To add a new message to the private queue named queue3\$ use the following code. There is a String label attached to each message and as you can see in Figure 9-5 the label is used by MMC and could be useful to applications that need to loop through the contents of a queue looking for specific messages or types of message rather than retrieving them all in sequence. To ensure transactional integrity of the changes made to the message queue by the Send method call, a MessageQueueTransaction is employed.

```
String msg = "message";
String label = "label";
MessageQueueTransaction mqt =
new MessageQueueTransaction();
mqt.Begin();
try {
mq.Send(msg,label,MessageQueueTransactionType.Single);
mqt.Commit(); }
catch(Exception ex) {
mqt.Abort(); }
```

Notice the third parameter to the Send method which specifies MessageQueueTransactionType.Single as the transaction type. A Single transaction The other two values of this enumeration are Automatic and None. Transactions of type Single are known as internal transactions. Internal transactions created by MSMQ can't be exported to or shared with other transactional resource managers. Automatic is the transaction type that must be used by applications that integrate MSMQ with DTC. Figure 9-5 shows the private queue named queue3\$ as viewed from the MSMQ management

console found in the Computer Management administrative tool. The sample shown here assigns each message the same generic label, and this label appears as the text identifier in each message line.

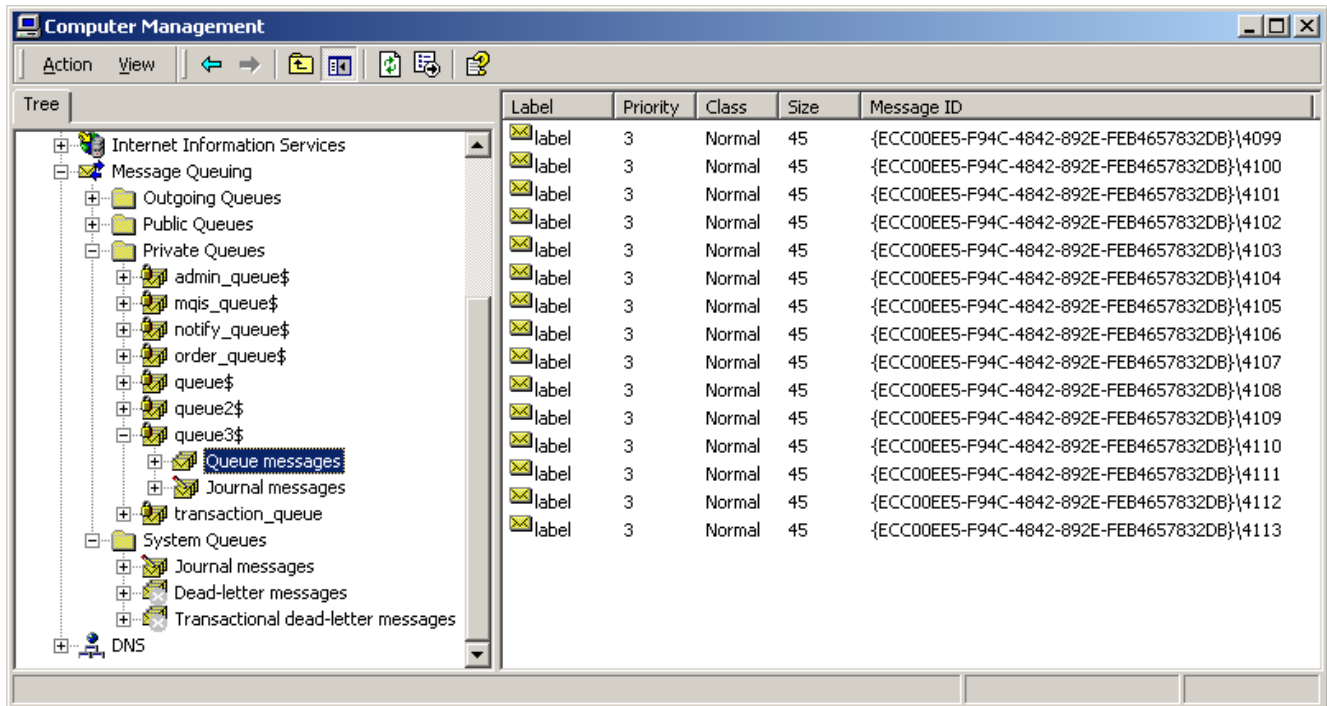


Figure 9-5: Use the MSMQ MMC to manage both local and remote queues

To retrieve messages from the queue within the context of a transaction, the following code is used to create another MessageQueueTransaction object that will manage the pending change (removal from the queue of a message or messages) and either commit the change or abort it as instructed. Like the previous code for sending a message to the queue in a transaction, the Begin method is called before any queue activities are performed and Commit is called when the work is done. The Receive method retrieves the next message in the queue.

```

MessageQueueTransaction mqt = new MessageQueueTransaction();
mqt.Begin();
try {
Message[] msgs = mq.GetAllMessages();
if(msgs.Length > 0) {
Message m =
mq.Receive(MessageQueueTransactionType.Single);
mqt.Commit(); }}
catch(Exception ex) {
mqt.Abort(); }

```

There are many features and configuration or usage scenarios possible with MSMQ. The most interesting among them for secure transaction processing include the ability to encrypt and digitally sign messages that are sent to queues and the ability to receive delivery confirmation of messages with help from the Dead Letter Queue and recoverable

messaging. The receiver can decrypt and automatically verify the signature applied to each message. Registry settings enable MSMQ-wide default settings for digital signature key pair management and other configurable security options. Properties of each queue can be viewed and altered using the MSMQ MMC shown in Figure 9-5. As securable objects, each queue has a different set of ACLs that restrict access and read or write privileges based on current security context.

Using a Response Queue with a Reverse Proxy Server

A thick client can control the transaction context explicitly and know with certainty that a transaction is committed before further actions are taken. A thin client Web browser will only know that a transaction has committed when it receives a complete HTTP response containing the result of a transaction it initiated. A reverse proxy can buffer in full the response sent to the client in order to avoid even a single preventable request processing failure. The reverse proxy can keep a transactional response, one that must be delivered to a client before the client makes additional requests in order to preserve an important sequence of events, buffered in the event of communications failure with the client that results in the client re-sending a duplicate request. Rather than having Web developers code duplicate request detection when such detection is important in order to prevent duplicate transactions, the reverse proxy can serve this purpose. IIS can be used as the foundation for such a reverse proxy server simply by searching a transactional message queue for a response to a particular request that is already pending before entering a busy wait state monitoring the response queue so long as the client connection is still active. Request processing, and responsibility for producing queued responses, is handed off to another server or process while IIS retains only the responsibility for delivering responses to clients. There's no reason in particular that an important response that a system tried to send previously to a Web client can't be resent to that same client in response to the next request it receives regardless of what that request asks the server to do. However, be careful to keep the user interface understandable and above all make sure you can reliably authenticate the client session as described previously in this book. Chapter 12 explains IIS-based authentication options in more detail.

Each participant in a transaction must comply with ACID requirements and conform to the same rules enforced by the transaction coordinator. Stand-alone transactional resources like relational database management servers (RDBMS) that communicate directly with client programs or application servers to initiate transactions don't usually give programmers and administrators the ability to define arbitrarily complex security policies to harden against attacks and penetrations. In practice, such transactional resources are data integrity smart and security dumb. A user ID and password combined with encryption through IPsec or database driver protocol settings are the extent of the security afforded by the typical RDBMS. When distributed transactions are supported by a transactional resource such as an RDBMS, On-Line Transaction Processing (OLTP) becomes DOLT (Distributed On-Line Transaction) Processing. DOLT Processing without measures to ensure security is just plain stupid. There is no good reason to allow distributed remote control of a transaction coordinator without reliable and trustworthy countermeasures to prevent data corruption, loss, theft, and denial of service in the event of malicious impersonations through credential theft or brute force password cracking.

An attacker who has control over a server that is granted permission to read and modify a transactional database or an attacker in possession of a valid user ID and password for the database server will never be locked out of a database automatically based on security policy violations detected by the database server itself. This is the most important reason to prevent direct access to a database server or similar resource and shield the resource through an application server interface such as IIS instead. The application server layer can enforce any security policy deemed necessary regardless of complexity. IIS provide the preferred platform under Windows server operating systems for building such security policy layers around transactional databases through the creation of transactional XML Web Services.

Chapter 10: Code Safety Assurance

Security is like a canoe: it's easily carried by only two people but every person third and beyond who tries to help carry the canoe just makes it more difficult. Terrorists know this so they tend to work in cells (at least until law enforcement puts them in one) rather than in the form of hierarchical monolithic organizations. Even peaceful political activists often work together in affinity groups to organize and execute protests or civil disobedience. Commanding a vast army from the top of the org chart means ensuring security, carrying that canoe, with the help of large numbers of people. This provably does not work and security in such situations is always a matter of damage control and the element of surprise rather than a provable absolute. In the end, all code is untrustworthy because human programmers are imperfect. A significant increase in practical data security can be achieved by reducing the number of programmers who carry the security burden.

Microsoft's core strategy along these lines is the migration of nearly every Windows program to .NET managed code. This enables a small number of highly-skilled infosec programmers working for and consulting with Microsoft to make it easier for everyone else who writes code to inherit, and benefit from, a trustworthy computing foundation.

A precept underlying all information security is that of containment through privileges and permission sets. The idea that an information system or a particular security principal, under normal circumstances, is only able to damage that which it is explicitly authorized to access. Denying IUSR_MachineName and other IIS impersonation accounts access to everything non-essential and configuring IIS to host applications outside of the inetinfo.exe process under a security context that is not highly privileged are the most important steps to lockdown IIS along these lines. But what of the abnormal circumstances? Any code that hasn't been proven secure is untrustworthy. The only failsafe protection possible in the event that untrustworthy code attempts something malicious is limited privileges and restrictive permission sets for the user account security context over which the code has control. It's important to ensure that every Web site and every application hosted under IIS operates under its own unprivileged user account and security context because of the possibility that malicious code will execute by way of hosting within or hijacking of a process spawned by IIS. This is challenging to accomplish in practice, however, because it requires a negative proof: that a particular user account in whose security context a process executes is in fact unprivileged.

Unanticipated privilege elevation can occur as a result of bugs in Windows OS code or standard features of its SDKs and APIs that, by design, enable such elevation for specific reasons. Privilege elevation can also occur through simple configuration changes in the Registry and elsewhere, requiring that all such configuration settings and storages be completely protected themselves. These interconnected dependencies that all must be comprehensively hardened and assurance tested in order to supply such a negative proof may be finite but they are not well-documented. This leads right back to the original precept, that if a highly-privileged security context wishes to do so it can ruin arbitrary aspects of privilege containment for any that is less-privileged. Security assurance for a user account under which IIS-hosted code is deployed, and the risk containment possible when deploying any such code, depends first on a presumption of cooperation by highly-

privileged code (as well as privileged user accounts) and depends next on the prevention of unauthorized, unplanned privilege elevation.

Software should never be controlled automatically or hosted by other software without rigorous security assurance beforehand in both the software and its host. Compiled code, in particular, carries with it numerous dangers when hosted or invoked by other software without human intervention. It's important to differentiate between the security risk inherent to compiled code deployed to an IIS box and the risk created by Webmasters and content authors who are confined to scripting of existing software services. Script developers, in general, pose a lower security risk for Internet Information Services than do programmers who create and deploy compiled code. Security policy and permissions settings combined with a thorough lockdown of production servers can effectively contain most coding risks posed by scripts that are interpreted by a script engine whereas compiled code executed directly by the microprocessor within a host process can be adequately secured only through rigorous source code review and special failsafe security countermeasures.

Countermanding Untrustworthy Code

Even though application code may appear to be safe right now, over time new vulnerabilities are discovered in other code and in a host platform such as IIS that can result in exploits designed to misappropriate innocuous software. Sometimes, as an attacker is searching for a way to turn a buffer overflow discovery into a workable exploit, the most important factor that determines whether or not a workable exploit can be designed is the set of executable instructions that happen to already be loaded into memory by the vulnerable process. The real questions are what does safe code look like and how do you ensure that your programming efforts or the servers you administer are not plagued by constant problems due to unsafe code? What can be done today to make it unnecessary to provide proof of a negative, that a particular security context is now and always will be entirely unprivileged? How can you make presence of unsafe code irrelevant to final analysis of whether an IIS box is secure? Much research and development still needs to be done to provide comprehensive answers to these questions and technical improvements to back them up.

Profiling and Blocking Application Initialization

Some day the right balance between paranoia and practicality will be achieved in the hardware and software with which we compute. In the mean time, the safest code is code that cannot be executed. Minimize the extent to which everything is potentially executable and you curtail the uncontrolled spread of code, giving you a chance, perhaps, to review some or all of it before it is allowed to execute. One way to accomplish this is to explicitly countermand the attempted execution of code that has not been reviewed for safety. A registry value called Applnit_DLLs makes this easy by injecting any DLL listed into any process that loads User32.dll into memory. This doesn't prevent all code from executing, since the DLL_PROCESS_ATTACH event occurs and DllMain is called for every DLL loaded by the process prior to the injected DLL, malicious code could still do harm by providing a DLL of its own or by avoiding loading User32.dll into the malicious process. Most applications link with User32.dll because of the Win32 API functions that live inside this system DLL.

Microsoft Knowledge Base article Q197571 gives more information about using the Applnit_DLLs registry value to inject a countermand DLL into every process.

Registry path: HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\Applnit_DLLs

The call to the main entry point function address of the code that starts a new process is delayed until all DLLs, including the injected Applnit_DLLs, are loaded. This means that an executable program doesn't get a chance to execute any of its logic if a DLL loaded into its process space calls TerminateProcess inside its DllMain function during DLL_PROCESS_ATTACH. To avoid problems caused by unpredictable DLL load order sequence and deadlocks due to reentrancy, DllMain is normally designed such that its startup code when a process first loads the DLL into memory calls only functions found in Kernel32.dll. Calling other external functions from within DLL_PROCESS_ATTACH is unsafe especially if the DLL is meant to be loaded into many processes or used by a third-party developer. Luckily, the Win32 API function TerminateProcess lives in Kernel32.dll and it can therefore be used safely and reliably to shut down a process that is denied permission to execute. The following code can be built as a DLL and installed by way of the Applnit_DLLs registry value in order to profile and countermand untrustworthy executables.

```
BOOL WINAPI DllMain(HANDLE hModule, DWORD ul_reason_for_call, LPVOID
    lpReserved) {
    bool bProfileMode = false;
    bool bBlocked = false;
    UINT uiSP = 0, ui = 0;
    DWORD dwBytes = 0, dwCreationDisposition = 0;
    HANDLE hProfile = INVALID_HANDLE_VALUE;
    char sProfileDirPath[MAX_PATH + 1];
    sProfileDirPath[MAX_PATH] = NULL;
    char sProfileModePath[MAX_PATH + 1];
    sProfileModePath[MAX_PATH] = NULL;
    char sProfiledPath[MAX_PATH + 1];
    sProfiledPath[MAX_PATH] = NULL;
    char sBlockedPath[MAX_PATH + 1];
    sBlockedPath[MAX_PATH] = NULL;
    char sSystemPath[MAX_PATH + 1];
    sSystemPath[MAX_PATH] = NULL;
    char buf[MAX_PATH + 1];
    buf[MAX_PATH] = NULL;
    if(ul_reason_for_call == DLL_PROCESS_ATTACH) {
        uiSP = GetSystemDirectory(sSystemPath,MAX_PATH);
        // 27 characters plus null: "\\ApplnitProfile\ProfileMode"
        if(uiSP > 0 && (uiSP + 28) < MAX_PATH) {
            if(lstrcpys(sProfileDirPath,sSystemPath) != NULL) {
                if(lstrcat(sProfileDirPath,"\\ApplnitProfile\\") !=NULL){
                    if(lstrcpys(sProfileModePath,sProfileDirPath) != NULL) {
                        if(lstrcat(sProfileModePath,"ProfileMode") != NULL) {
```

```

hProfile          =          CreateFile(sProfileModePath,GENERIC_READ,
    FILE_SHARE_READ,NULL,OPEN_EXISTING,NULL,NULL);
if(hProfile != INVALID_HANDLE_VALUE) {
bProfileMode = true;
CloseHandle(hProfile);
hProfile = INVALID_HANDLE_VALUE; }
dwBytes = GetModuleFileName(NULL,buf,MAX_PATH);
if(dwBytes > 0) {
// path may be UNC path "\\?\*" or local path "C:\*"
// replace each backslash or colon with underscore
for(ui = 0;ui < dwBytes;ui++) {
if(buf[ui] == '\\' || buf[ui] == ':') {
buf[ui] = '_'; }}
// 8 characters plus null: "BLOCKED "
if(strlen(sProfileDirPath) + dwBytes + 9 < MAX_PATH) {
if(lstrcpy(sBlockedPath,sProfileDirPath) != NULL) {
if(lstrcat(sBlockedPath,"BLOCKED ") != NULL) {
if(lstrcat(sBlockedPath,buf) != NULL) {
hProfile = CreateFile(sBlockedPath,GENERIC_READ,
FILE_SHARE_READ,NULL,OPEN_EXISTING,NULL,NULL);
if(hProfile != INVALID_HANDLE_VALUE) {
CloseHandle(hProfile);
bBlocked = true; }}}}
if(!bBlocked) {
if(bProfileMode) { // Profile Mode
dwCreationDisposition = OPEN_ALWAYS; }
else { // Protect Mode
dwCreationDisposition = OPEN_EXISTING; }
if(lstrcpy(sProfiledPath,sProfileDirPath) != NULL) {
if(dwBytes + strlen(sProfiledPath) < MAX_PATH) {
if(lstrcat(sProfiledPath,buf) != NULL) {
hProfile = CreateFile(sProfiledPath,GENERIC_READ,
FILE_SHARE_READ,NULL,dwCreationDisposition,NULL,NULL);
if(hProfile != INVALID_HANDLE_VALUE) {
CloseHandle(hProfile); }
else if(!bProfileMode) {
hProfile = CreateFile(sBlockedPath,GENERIC_READ,
FILE_SHARE_READ,NULL,OPEN_ALWAYS,NULL,NULL);
if(hProfile != INVALID_HANDLE_VALUE) {
CloseHandle(hProfile);
bBlocked = true; }}}}
if(bBlocked && !bProfileMode) {
TerminateProcess(GetCurrentProcess(),0); }
return TRUE; }

```

Inside the DLL_PROCESS_ATTACH code of User32.dll's DllMain function is a conditional call to LoadLibrary that loads each DLL listed under Applnit_DLLs. This means the only DLL function that will be called in this countermand Applnit_DLLs library is DllMain since no code inside the host process into which this library is injected has any knowledge of

the presence of this countermand DLL. To install the DLL, first build it with a compiler. Next, issue the following commands at the command prompt using an administrator security context to create the ApplnitProfile directory under System32 into which the countermand library DLL writes profile information and logs blocked executables. The ProfileMode file indicates to the DLL that it should not TerminateProcess but rather profile executable invocation only, giving you a chance to run through a typical day or week of computer operation before switching to protect mode. As long as there is a file named ProfileMode in the ApplnitProfile directory, the DLL remains in profile mode.

```
MKDIR %SYSTEMROOT%\System32\ApplnitProfile  
TIME /T >%SYSTEMROOT%\System32\ApplnitProfile\ProfileMode
```

An updated version of the countermand Applnit_DLLs library source will be published periodically at its open source project Web site: <http://www.countermand.org>

Figure 10-1 shows the contents of the ApplnitProfile directory after a period of time has elapsed operating this countermand library in both profile mode and protect mode. Notice the BLOCKED entry for WScript.exe, the Windows Script Host (WSH). Many deployments choose to prevent the use of WSH by removing the executable from the system but encounter difficulty preventing its reintroduction by service packs or authorized users. The DLL source code to implement this application initialization countermand defense is simple to understand, and the code as shown is hardened against potential vulnerabilities through explicit buffer length verification, pessimistic string null terminator redundancy and by virtue of the fact that the parameter values passed in to theDllMain function have no impact on the code except to indicate ul_reason_for_call. The only way the source code shown here could be controlled by malicious input is through interception of the Win32 API calls into Kernel32.dll for string manipulation, file creation, and TerminateProcess. Beware, however, that any runtime library code you allow your linker to include may introduce flawed prolog and epilog code, although exploitation even of that code is unlikely considering the only code that ever calls the Applnit_DLLs is LoadLibrary.

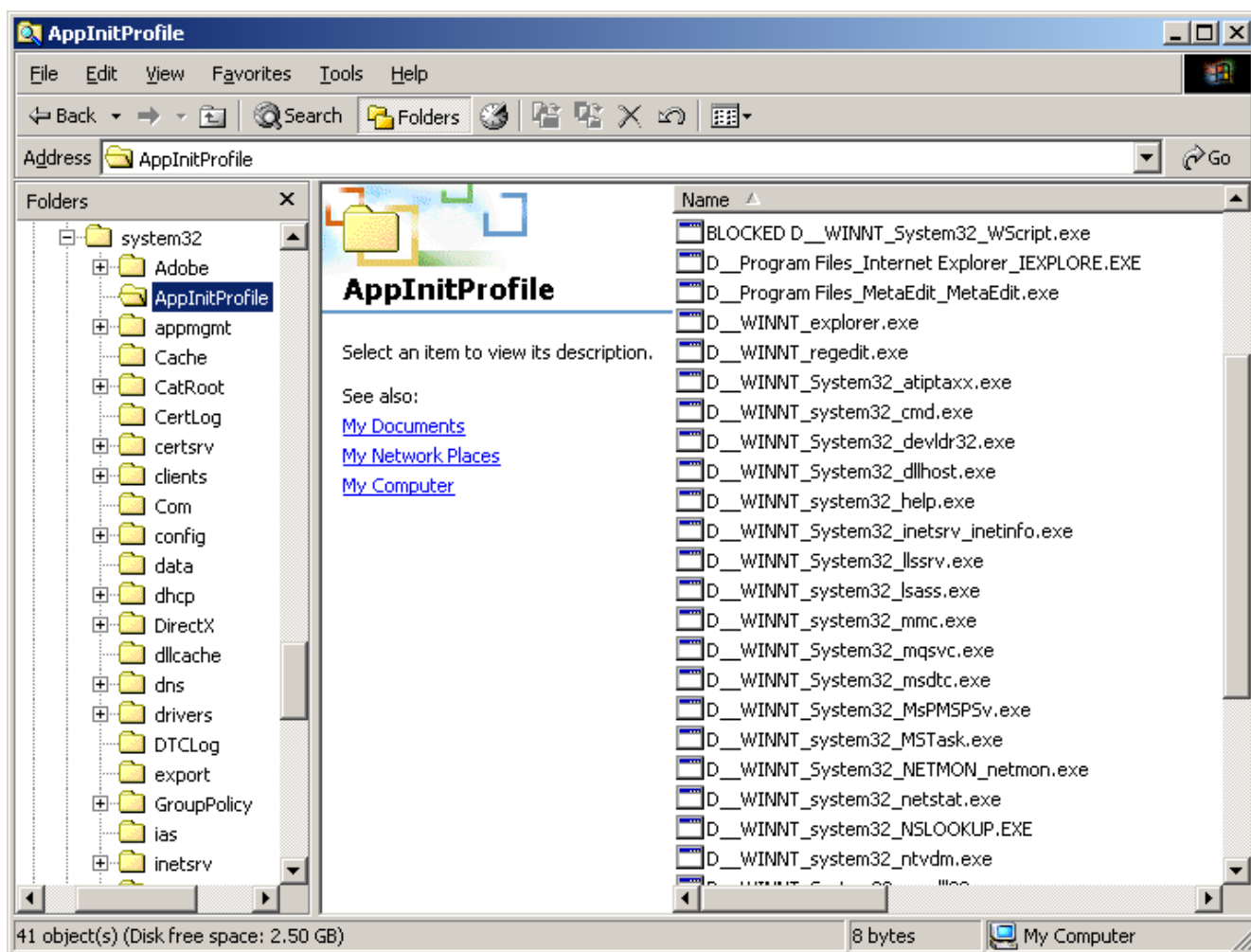


Figure 10-1: A Countermand DLL Can Profile and Block Most Executables

After setting up local variables and adding extra null terminators at the end of each MAX_PATH character buffer, the code attempts to open the ProfileMode file located in the AppInitProfile directory. If the file is present, the countermand DLL operates in profile mode and never calls TerminateProcess. Next a call to GetModuleFileName with the first parameter NULL gives us the full path to the executable module responsible for creating the current process. This path might be an UNC share or a local drive letter. All backslashes and colons are replaced with underscores in the resulting character buffer and the prefix "BLOCKED " is prepended to come up with the filename of the file that, if present in AppInitProfile, indicates that the executable module path for the current process matches one that has been blocked. In this case there is no further need to profile the module and the decision to block the process by calling TerminateProcess has already been made if the countermand library isn't operating in ProfileMode. When there is no existing BLOCKED file found in AppInitProfile, the code attempts to open a file without this prefix. In ProfileMode, the file is opened or created if it doesn't exist resulting in a new file named with underscores in place of backslashes and colons to memorialize the module's execution. In protect mode, the absence of a file without the BLOCKED prefix indicates that the executable has never been profiled, and it is therefore blocked by default and a new BLOCKED file is created to record this fact. When put into production use, any executable module that is blocked by the countermand library can be authorized

for future execution simply by removing the "BLOCKED " prefix from the filename of the empty file created in the ApplnitProfile directory.

One note of caution is required if you plan to deploy the Applnit_DLLs countermand library to block unauthorized program execution. Because of the immediate process termination that occurs when a process is blocked by the library there is never any notification sent to other DLLs that may already have performed their DLL_PROCESS_ATTACH initialization. Some DLLs are designed to hold values in shared memory blocks allocated dynamically and there is a chance of memory leaks occurring in such DLLs due to the fact that they are never given the corresponding DLL_PROCESS_DETACH notification. However, experienced programmers who build such DLLs should know better than to rely on DLL_PROCESS_DETACH as the sole means of triggering garbage collection of resources allocated dynamically in shared memory. Many DLL programmers delay any such shared memory allocations until they're actually needed rather than performing these allocations insideDllMain. Still, memory leaks aren't as important to prevent completely as are unauthorized executable programs. An increased risk of memory leaks (and therefore DoS) is usually an acceptable trade-off in this respect.

Restricting NTFS ACL Execute Permissions

DACLs include a Read & Execute permission that can be used to prevent execution of any file stored on an NTFS filesystem. It's useful to understand where this execute restriction comes from, what it means, and when it applies. When a 32-bit Windows portable executable (PE) formatted program file containing executable machine code instructions for the appropriate target microprocessor architecture (x86, DEC Alpha, or various processors under Windows CE) is executed, it is presented to the Windows loader which calls CreateProcess after loading the machine code instructions into RAM. The executable image stored in the PE-formatted file includes the initial entry-point for the main thread of execution and it also includes the root stack frame from whence the call stack grows as the thread of execution continues. Whether or not a particular file is treated as an executable by the windows loader, and therefore whether or not it searches for PE-formatted data within the file to load into RAM prior to calling CreateProcess is determined by a simple list of registered file types that map file extensions to applications that handle events, such as open or edit, for each type of file. A special file type association called Application exists implicitly by default on every Windows box and doesn't appear in the Registered File Types list box shown in Figure 10-2 for files with .EXE but it exists nonetheless. To access this list box you run Windows Explorer and choose Folder Options from the Tools menu then click on the File Types tab.

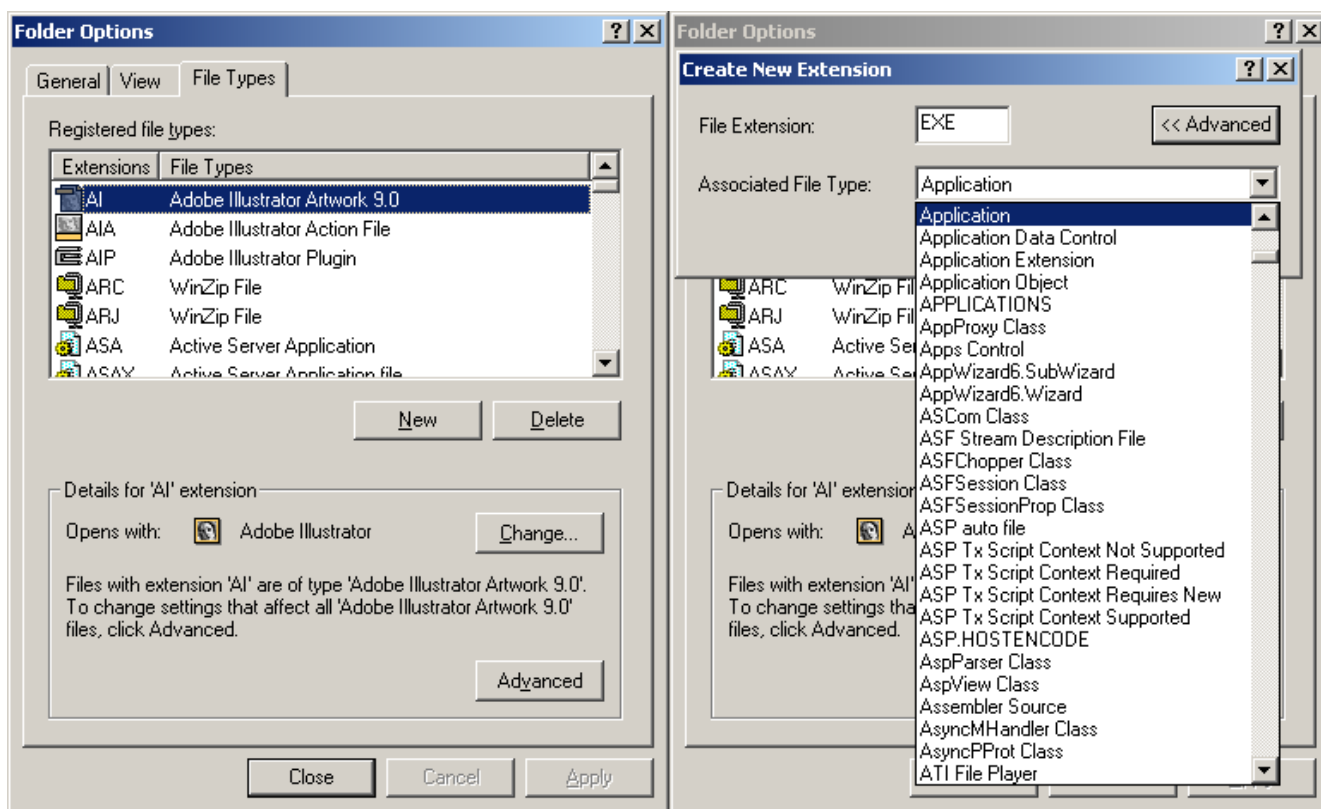


Figure 10-2: Application Type File Extensions Enable Program Execution

You can modify the .EXE file extension type mapping explicitly by clicking the New button, typing EXE in the File Extension field, and selecting from the drop-down list box labeled Associated File Type as shown in Figure 10-2. Doing so produces a warning about the .EXE file type already being associated with Application. Confirm the replacement and your new .EXE file type appears in the list and you can add event handlers through the Advanced button. File extension mappings are stored in the registry under HKEY_CLASSES_ROOT and you can look there using the Registry Editor to see the default file extension mapping for executable applications. MIME types are also configured for Internet Explorer and other Windows applications that produce or consume data based on its MIME type and file extension. The MIME type of each file is listed in HKEY_CLASSES_ROOT under a REG_SZ value named Content Type. Changing this value for a particular file extension does not alter the Content-Type HTTP header produced by IIS when they deliver data from files with particular file extensions because IIS have MIME type mappings stored in the metabase for this purpose. The following metabase path is where MIME types are stored for Content-Type headers sent in HTTP responses.

\LM\MimeMap

The SubSeven Trojans and many other types of malicious code change registry settings that impact the ability of the shell to execute .EXE files unless they are first renamed with a different file extension that is still associated cleanly with the Application file type, such as the .COM file extension. You can replicate this exploit condition manually by creating a new file type setting for .EXE, replacing the Application type mapping, and then deleting the new entry from the Registered File Types list box. The new .EXE entry appears in the

list until you close and reopen the Folder Options window, so you can select it and press the Delete button. Then navigate to an .EXE file in the Windows Explorer and double-click on it to see what happens: you can no longer execute programs in .EXE files. Return your system to normal by repeating the Create New Extension process as shown in Figure 10-2.

For file types that are not executable in the view of the Windows loader, a handler program must be executed instead that is designed to make use of the data contained in the file. For such handler programs the loader assumes that the file containing the handler program is of type Application, which gives rise to the possibility for all Application type file extensions to be associated instead with a handler program that provides an extra security layer to control program execution and implement security policy. To see how this would work, associate the .EXE file extension with notepad.exe as the handler for the Open event. Now, every program that used to be executable (other than handler programs) instead opens notepad which attempts to display the contents of the executable file as a text document. Carefully undo this setting so that .EXE is properly associated with the special Application type before you reboot otherwise your services and other executable code will fail and your Windows box will probably refuse to boot properly. Such a security policy handler wrapper around executable code would remain in force unless the registry was modified to remove the handler or set another file extension as Application type.

In addition to the Application type mapping for executable files based on the .EXE file extension, DACL security permissions on the HKEY_CLASSES_ROOT registry key for .EXE files also control which user security contexts are allowed to access files of this type. If read access is not granted on this registry key to the user security context in question then the Windows loader fails to open the key and is unable to determine that the file is an Application type that should be treated as potentially executable. As a result, the user is unable to execute programs. This is a good way to explore the subtleties of the DACL and the meaning and impact of the Execute permission under Windows. Figure 10-3 shows the Advanced security settings for a registry key. Understanding the variety of DACL permissions that interact to determine whether or not a particular executable module, DLL, COM class, or data file can be used for a particular purpose is an important part of ensuring security of IIS and applications hosted by IIS.

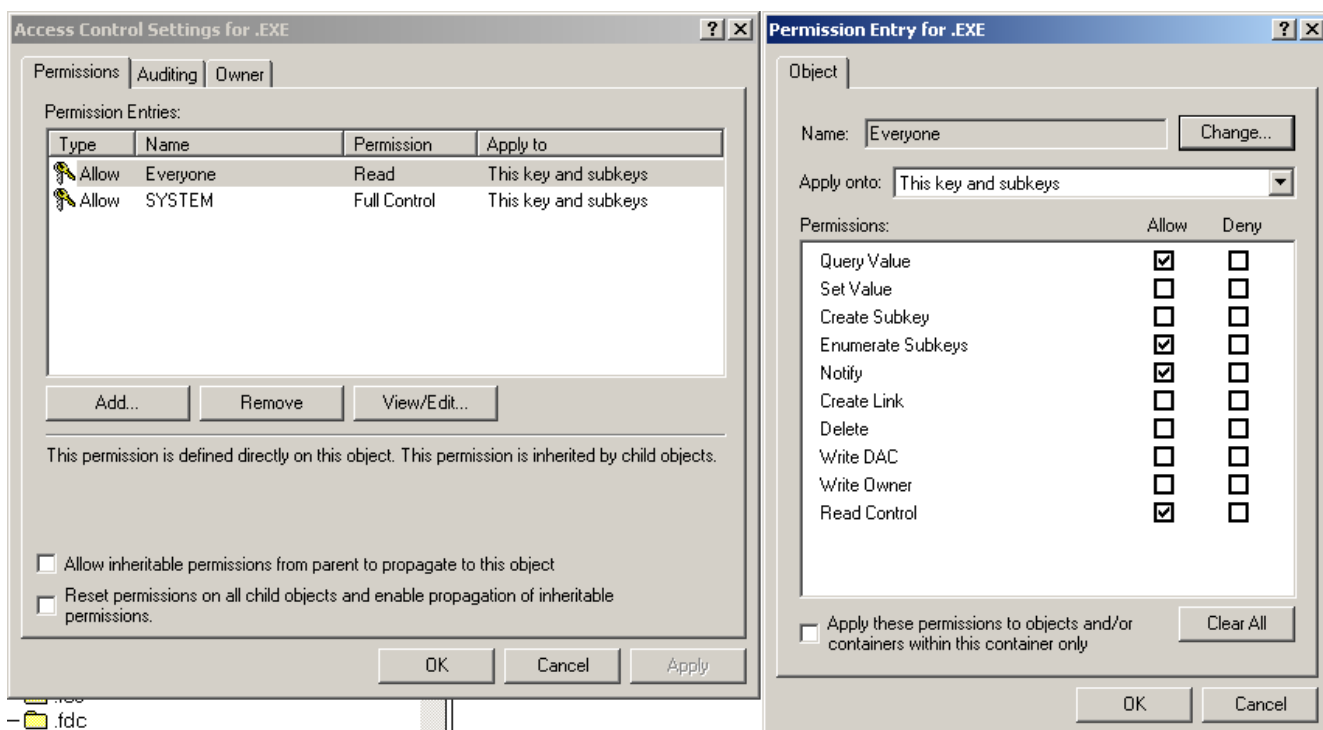


Figure 10-3: Setting Advanced DACL Permissions on The .EXE Registry Key

A lengthy process of hardening and locking down any IIS box is necessary before it is put into production. The majority of this lockdown work involves documenting and verifying permissions settings, DACLs, on everything. All folders and files, all registry keys and values, and every metabase entry has a DACL that must be documented and verified. Documentation is especially important because it enables subsequent verification. The best way to manage this documentation is to use the Security Configuration Manager (SCM) system. SCM makes it possible to verify on-demand that security configuration on a box still complies with established policy.

The following Knowledge Base articles discuss the SCM in more depth:

Q245216 Downloading and Using the Security Configuration Manager Tool

Q214752 How to Add Custom Registry Settings to Security Configuration Editor

Q195509 Installing Security Configuration Manager from SP4 Changes Windows NT 4.0 ACL Editor

Q271071 Minimum NTFS Permissions Required for IIS 5.0 to Work

The National Security Agency publishes secure configuration default recommendations for Windows that make use of SCM. You can download the latest version of the NSA configuration default security recommendations in the form of security configuration templates for use with the SCM's Security Configuration Editor from the NSA Web site located at the following URL:

<http://www.nsa.gov>

Windows .NET Server and Windows XP establish greatly-improved default security settings compared to Windows NT and Windows 2000. A thorough review of default permissions is essential no matter what server OS version you use to run IIS. It's important to note

that the convert.exe utility that will migrate a drive to NTFS from FAT or FAT32 grants everyone full control to everything so it should never be used on the system drive in any version of Windows. The regular installation process must be used instead and a fresh NTFS filesystem should always be the target volume to which the OS installation is performed.

The following Knowledge Base articles list default DACL settings in Windows:

Q244600 Default NTFS Permissions in Windows 2000

Q148437 Default NTFS Permissions in Windows NT

The following Knowledge Base Articles detail dangers of default permissions:

Q327522 Windows Default Permissions Could Allow Trojan Horse Program

Q300691 HOW TO: Set Up a File System for Secure Access in Windows 2000

One area in which you may want to deviate from the defaults and even from NSA's recommendations is in the area of permissions for DLL files. Although not executable themselves as stand-alone modules, LoadLibrary enforces the DACL requirements for execute permission on any DLL that it attempts to load into the process space created by an executable module. It isn't possible often times to remove DLLs that you don't wish to trust otherwise the software that makes use of them won't function. For system DLLs protected by Windows File Protection, custom permissions are reset to defaults established by the folder permissions when WFP copies the file from the dllcache directory or a .cab file. Removing WFP-protected DLLs isn't possible, but it is possible to remove Execute permission from such DLLs. Figure 10-4 shows advanced DACL permissions for the ASP.DLL file without the Execute File permission needed in order for IIS to load this DLL into memory and execute the ASP script engine that this DLL contains.

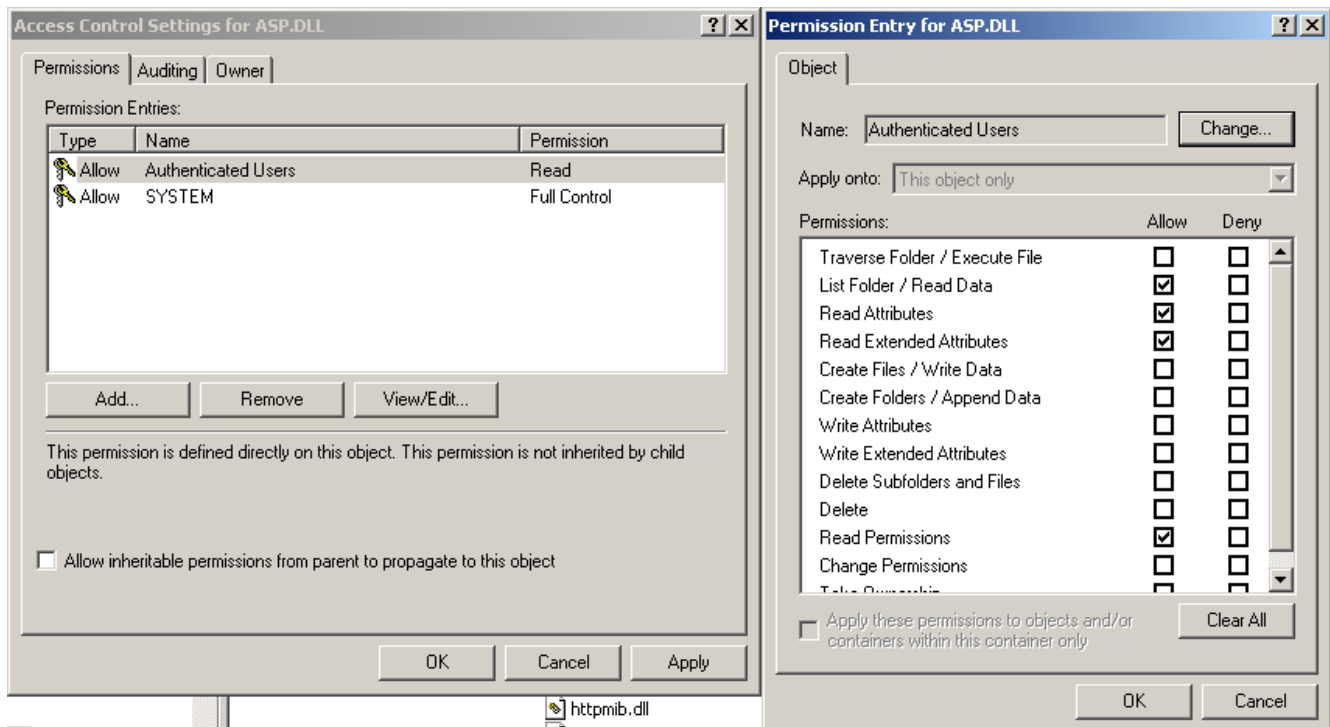


Figure 10-4: Setting Advanced DACL Permissions on ASP.DLL

With Execute permission removed on ASP.DLL there is no way for inetinfo.exe, mtx.exe or dllhost.exe to load this library into memory and therefore the ASP script engine is not accessible to any Web site hosted by IIS. This type of security assurance is especially important in cases where your IIS box hosts other people's code. In a case where the metabase settings for a Web site end up being modifiable by the webmaster of that site this DACL setting makes it harder to install the ASP script engine for use by the site even if new ISAPI extension mappings are added to the metabase. This is especially important in cases where co-workers have physical access to an IIS box and can edit the metabase for sites they program or administer but should never have the right to put into production a script engine that your security policy chooses to prevent from executing. ASP.DLL is one such script engine unless you have a need to support the hosting of legacy code. All new systems should switch to ASP.NET instead due to its improved security over classic ASP.

Script poses special dangers for Windows because of the split between the notion of executable code and non-executable data that requires a handler program. The HKEY_CLASSES_ROOT registry settings automate the execution of handler programs based on file extension, and this automation constantly results in new vulnerabilities. In addition, there is currently no way for a DACL to specify script execution permission differently than program execution permission, and the typical script engine fails to recognize the security improvement of requiring Execute File permission in the DACL in order to allow interpretation of a particular script by the script engine. As a result, every data file is potentially executable if it contains script and can be fed to a script engine that will interpret the script without regard for the Execute File DACL permission setting. This vulnerable script engine design should be remediated and future releases of Windows should add Interpret Script to DACL.

Compiler Security Optimizations

Nearly every software vendor in the last twenty years has faithfully repeated the same information security mistakes made by every other software vendor before them. Standard operating procedure throughout the twentieth century for software vendors was build it, ship it, fix it. For the twenty-first century software vendors are expected to behave differently, and the toolset used by programmers today reflects this change. Security optimizations in runtime libraries and compilers are part of the new and improved software industry. Do they prevent security problems? No, but they're a lot of fun. And they do more good than harm, hopefully. One such compiler security optimization that has had a big impact on IIS version 6 is the new Visual C++ /GS compiler option. Most of IIS version 6 was written with Visual C++ and unlike previous releases of IIS, IIS source code is now compiled with the new Guard Stack (/GS) option in the Visual C++ compiler.

In Chapter 1 you saw the simplest possible buffer overflow, where an array index reference exceeds the dimensional boundary of a stack-allocated buffer and thereby modifies the function return address that is popped off the stack when the microprocessor encounters a return instruction. Most buffer overflow vulnerabilities exploitable in the wild involve less explicit means of stuffing malicious bytes into stack memory, so that every byte of memory beginning with the starting address of the buffer is overwritten with malicious bytes that have just the right length and structure to drop a replacement value onto the authentic return address. This messy slaughter of everything in between the memory

buffer and the function return address leaves evidence of a buffer overflow condition, if we could just get all attackers to give us predictable malicious bytes in their exploit byte stuffing code then we could detect such conditions at runtime. Or, we could do as coal miners did before technical safety innovations improved gas fume detection: bring a canary with us everywhere we go. The coal miner's canary died quickly in the presence of odorless toxic fumes, and when the canary died the miner knew to leave the mine while the fumes dissipate. Our electronic canary equivalent is a cookie, a token of unpredictable value selected at runtime that we can use to confirm that the authentic return address has not been modified before we allow the microprocessor to trust it and pop it off the stack. The /GS compiler option in Visual C++ 7.0 places just such a canary on the stack and checks to see that it is still alive when a vulnerable stack frame returns. A modified version of the Hello World! stack buffer overflow Chapter 1 sample appears below.

```
int main(int argc, char* argv[]) {
    void * p[2] = {(void *)p[3],(void *)p[4]};
    char unchecked[13];
    p[3] = (void *)&p[0];
    p[4] = (void *)0x00411DB6;
    printf(strcpy(unchecked,"Hello World!\n"));
    return 0; }
```

This code, when compiled with /GS enabled in Visual C++ 7.0, results in precisely the same endless loop demonstration as illustrated in Chapter 1 (where the compiler used was Visual C++ 6.0). The difference is that when p[4] is set equal to the address of the original call to the main function (to set up the infinite recursion while reusing the current stack frame base address) 0x00411DB6 it results in the death of the canary and program execution terminates abruptly after the first iteration. Figure 10-5 shows the security cookie being retrieved into the EAX register. This cookie value was generated dynamically by the C runtime prolog code and stored in memory location 0x00425B40 where the compiled code expects to find it at runtime. The security cookie is not the canary, it is the pseudorandom encryption key used to produce the canary through the very next instruction involving exclusive or (xor).

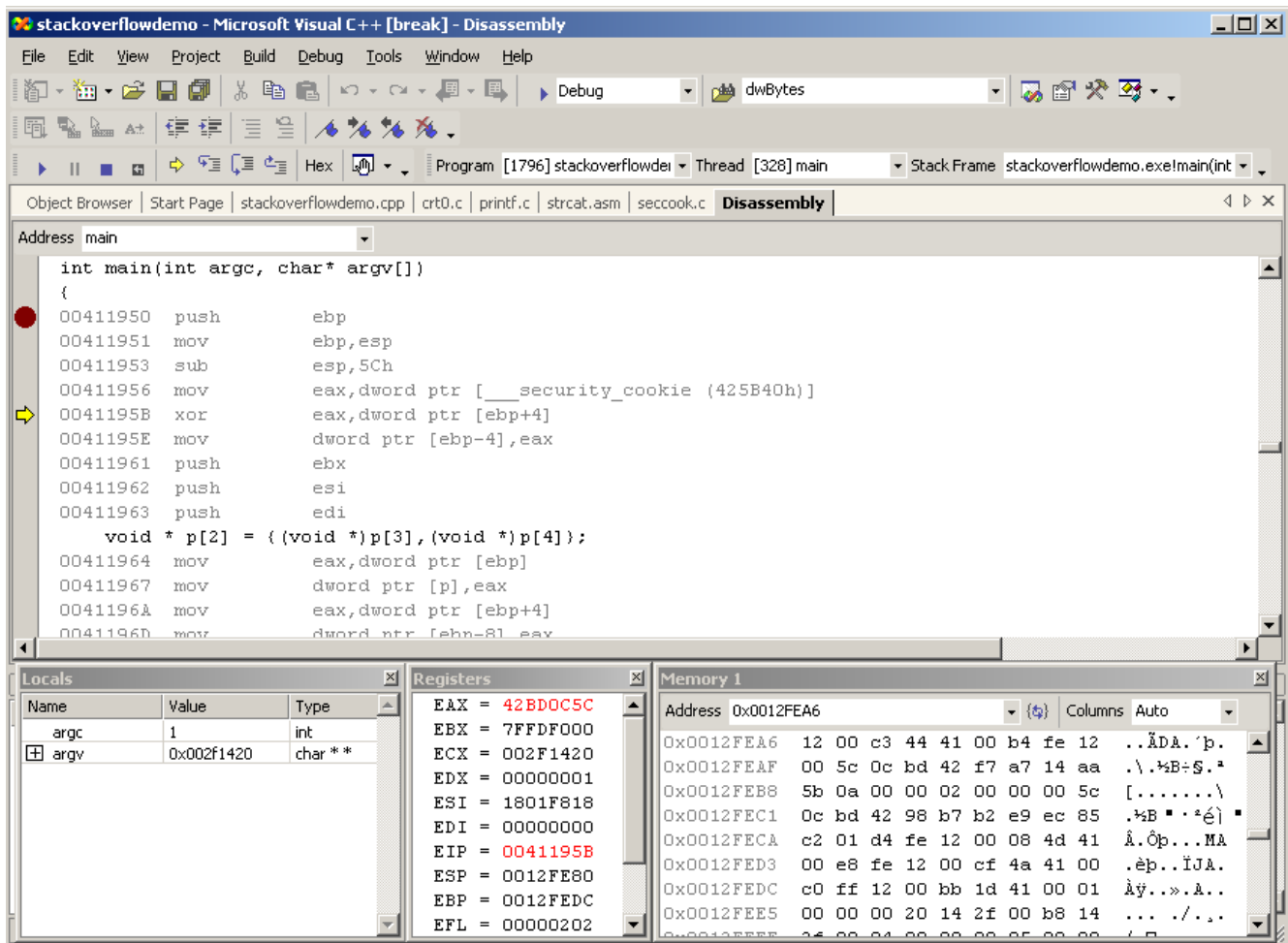


Figure 10-5: The Guard Stack Security Cookie Retrieved into EAX

What happens next is the birth of the canary, or its placement into the cage if you prefer to think of it in those terms. The canary is the bitwise xor combination of the security cookie and the authentic return address to which the current call stack expects program execution to return when the subroutine completes. Figure 10-6 shows the canary, the value 42FC11E7 stored in the four bytes just below the previous stack frame base address that was pushed onto the stack in the very first instruction at the beginning of the main function (`push ebp`). The four byte canary is placed in this location because it is the last four byte region of the new stack frame. Any buffer overflow exploits that impact the stack frame will have to write through the canary to get at the return address, which is stored in the four bytes of memory beginning four bytes above the base address of the new stack frame. Between the canary and the return address are the four bytes containing the previous stack frame base address 0012FFC0. The authentic return address shown in Figure 10-6 is 00411DBB which you can see on the line addressed beginning at 0x0012FEDC which happens to be the current value of EBP; the current stack frame base address.

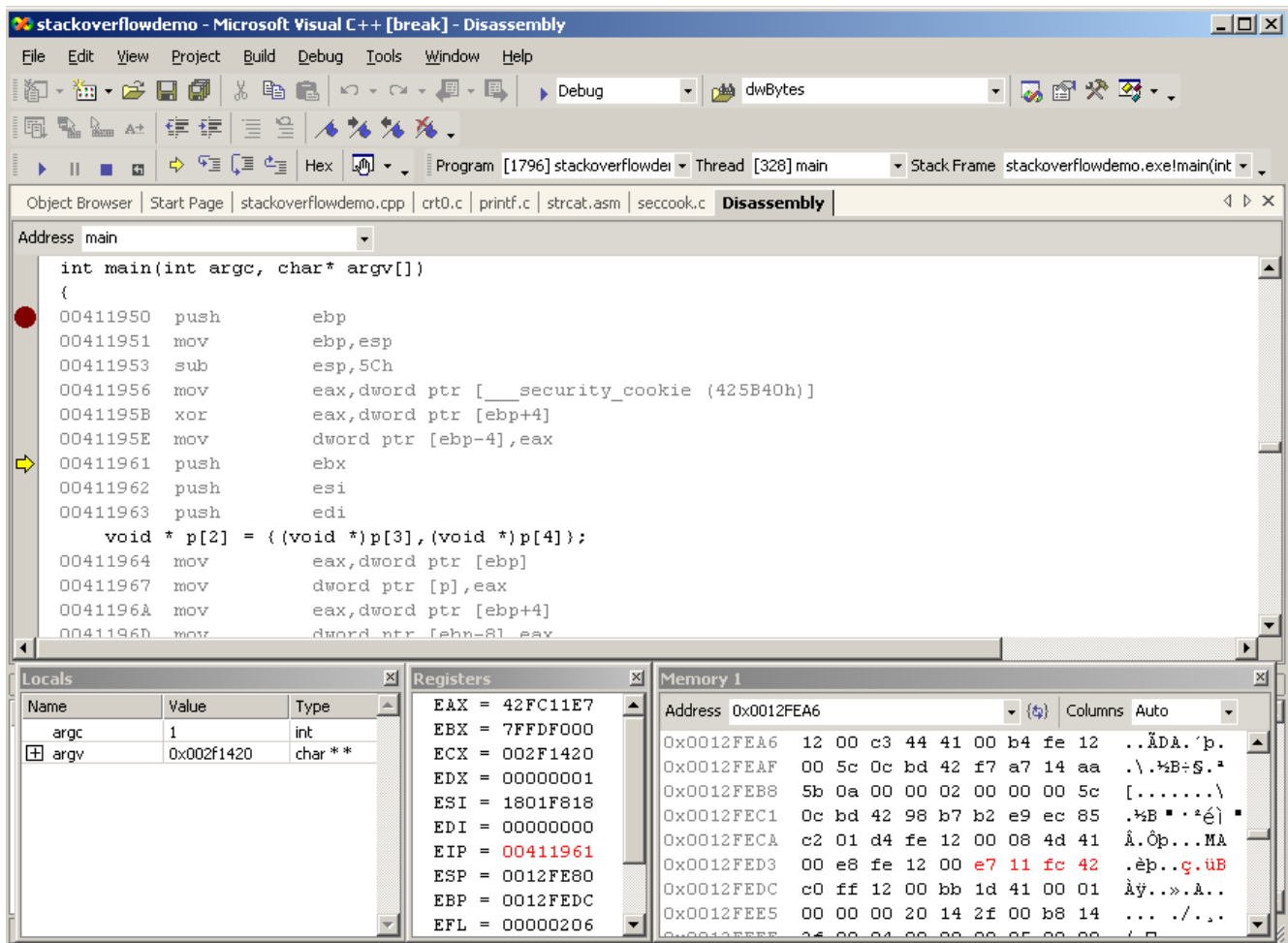


Figure 10-6: An Electronic Canary 42FC11E7 on The Stack

Like the sample shown in Chapter 1, this sample's mission is to capture the previous stack frame base address into the first element of void pointer array p and then take control of the return address to which the present stack frame will jump when the next return instruction is executed. Figure 10-7 shows these steps being carried out as planned. The hard-coded address 0x00411DB6 is the address to which the sample exploit prefers instead of the authentic return address and you can see the mov instruction at the top of the disassembly shown in Figure 10-7 that forces this new value in place of the authentic original. The authentic address of the previous stack frame base has also been overwritten by this time. Both of these malicious replacement values appear on the line addressed beginning at 0x0012FEDC. The next instruction to be executed, marked by the arrow and the address shown in EIP, moves the canary into the EXC register where it can be examined.

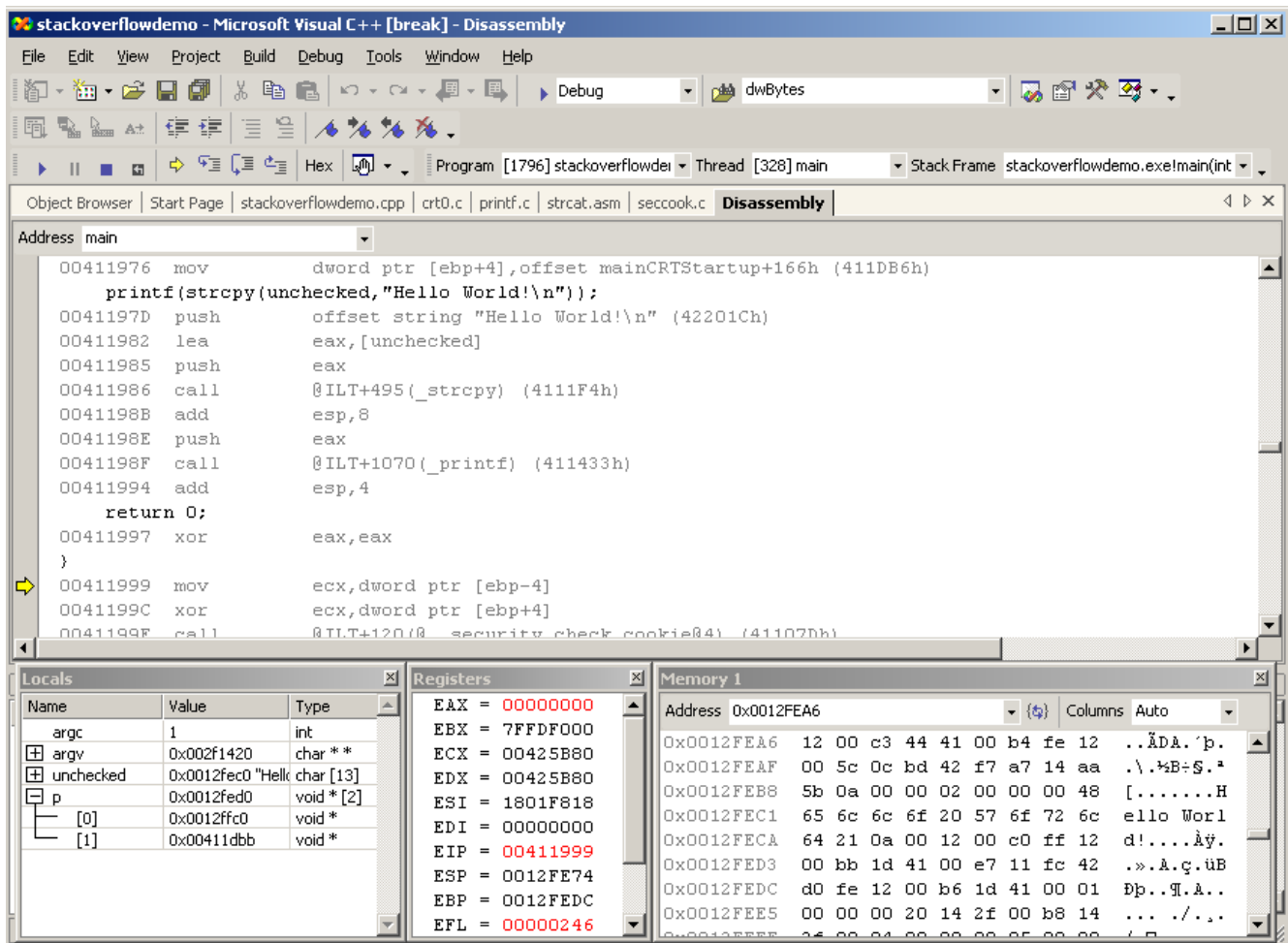


Figure 10-7: Our Sample Exploit Code Hard at Work

You can see in Figure 10-8 that the canary is still alive in its cage located at the very top of the stack frame where it was first placed as shown in Figure 10-6. Or is it? The canary value hasn't changed, but the return pointer has. Another quick xor using the new return address and the same security cookie as used previously and we can take the pulse of the canary to see if it's really alive. Figure 10-8 shows the result. The ECX register contains a value other than the security cookie and the canary is shown to have died. Of old age, perhaps, since it's now outdated and doesn't confirm the authenticity of the return address that program execution is about to be handed over to when the return instruction is encountered. The `__security_check_cookie` runtime library function calls `ExitProcess` when it detects the security compromise.

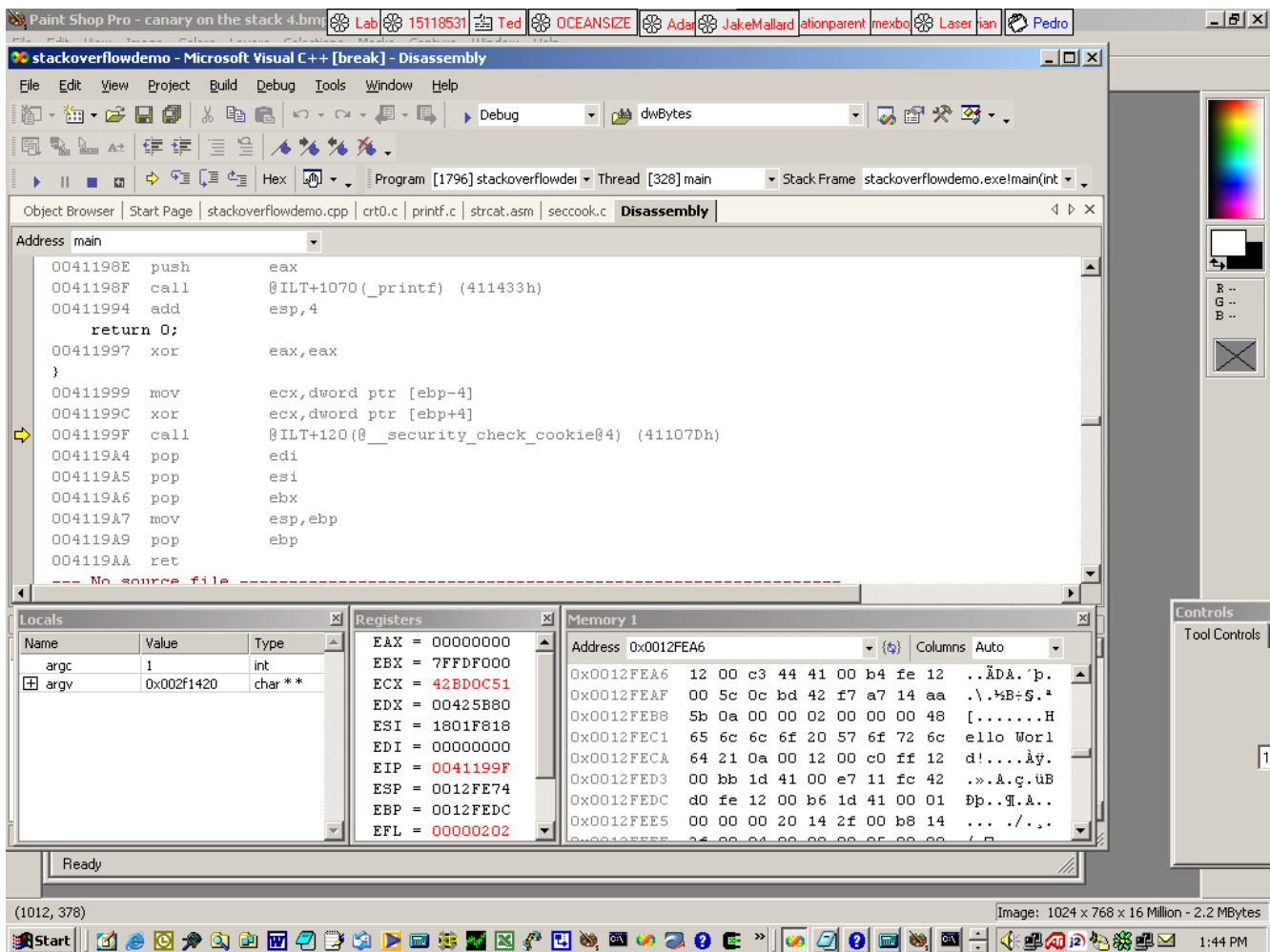


Figure 10-8: The Canary Looks Fine Until It Fails The Security Check

The compiler used to build object code from C++ source may have some security optimizations available but it can't change the basic fact that low-level control over code safety is placed in the hands of the C++ programmer. Many of the problems that are caused by unsafe code are indefensible so long as the unsafe code is present. Without a way to predict the type of problems that impact data security you can't add defensive layers around code that might benefit from such layers. Guarding against predictable problems is important, but if you already knew what all of the problems were and where those problems lived in code you would remove the dangerous code completely or place protective wrappers around every bit of it. This makes the unknown more dangerous than the known, and code known to be unsafe is often allowed to execute in spite of its attendant risks because the risks are known.

Modeling Vulnerable Code

The code safety assurance problem is two-fold. First there is review of your own source code and scripts for vulnerabilities. This requires an in-depth understanding of unsafe coding practices, something that can only be acquired through a review of the flaws that other people have created that resulted in exploitable security bugs. Next there is security quality assurance of other people's compiled code. This part is much more difficult.

Debugging tools are useful for this purpose, especially when the software vendor distributes a debug/checked build of their compiled code so that debugging tools can reveal more useful information about what a program does and why it does it. Calls to the `OutputDebugString` function are often left in debug builds, also, but removed from release builds. `OutputDebugString` is a Win32 API function that sends string output to a debugger if one is active on the system or attached to the process. Another important way to achieve security quality assurance of other people's code is to insist that they build and deploy COM+ interfaces and XML Web Services rather than APIs for any functionality exposed by the program that can be automatically invoked from other code. Remote Procedure Call (RPC) interfaces, network interfaces like Windows Sockets (Winsock), and DLL-based APIs usually offer only all-or-nothing control rather than fine-grained control over access to the services provided by the code. COM+ interfaces and XML Web Services, both of which can be built easily using .NET managed code, enable administrators to restrict access to certain interface methods but allow access to others.

It's very important to understand, whether you are a programmer or an administrator, that every line of code ever written, past, present, and future, is untrustworthy if it exposes legacy programmability interfaces that don't allow customized security access controls.

Exception Handler Function Pointer Replacement

Another common vulnerability caused by insecure compiler design is exception handler function pointer replacement. This is the exploit technique used by the Code Red worm and it's very easy to understand. When exception handling is enabled, a function pointer to the exception handler routine is stored on the stack in addition to everything else that is normally placed there. By stuffing a stack-based buffer with malicious bytes and overflowing that buffer all the way to the exception handler function pointer so that it is overwritten with a reference to a nearby address on the stack where the malicious bytes of the attacker's choosing were stored, the exploit setup is complete. All the attacker need do next, or immediately thereafter, is force an exception to occur so that the exception handler is called. This type of attack allowed Code Red to forcefully take control of the thread of execution rather than wait for a malicious return address pointer to be accepted passively. The /GS compiler optimization in Visual C++ 7.0 does not guard against exception handler function pointer replacement because the `__security_check_cookie` runtime library function is not called until the stack frame created by the current subroutine call is ready to return. Since an exception is forced before that time and execution transferred immediately to the exception handler, the death of the canary is ignored completely. To protect against the severity and immediacy of this exploit technique, Windows XP introduced an improved exception handling execution environment that prevents the exception handler function pointer from referencing stack memory. Additionally, Windows XP clears all registers before calling the exception handler function.

Suppressing Bad Memories

Any time a memory buffer is filled with bytes of unknown origin and those bytes fail a safety validation check, they must be purged from memory immediately. All sensitive information stored temporarily in memory must likewise be purged reliably when no longer in use. When script forms the basis of an application hosted by IIS, never assume

that the script engine will purposefully purge sensitive data from memory. It probably won't, and it probably will allow sensitive information to get swapped out to the system paging (swap) file. Unless scripts explicitly remove sensitive data from memory by overwriting values before the scripts terminate, there is a good chance that these sensitive values will stick around much longer than you expect. Compiler optimizations can also lead to the removal of instructions that the compiler deems unnecessary, such as a rash of seemingly-useless instructions that store values into memory that is never referenced again in the program. To a compiler these instructions might seem extraneous in the same way that a variable that is set to a value but never referenced serves no apparent purpose. For more on this topic see:

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncode/html/secure10102002.asp>

Chapter 3 detailed a technique in Asymmetric Cryptography for Bulk Encryption whereby bulk data encryption can be accomplished using asymmetric cryptography with a public key/private key pair rather than a single symmetric key for both encryption and decryption. One of the distinct advantages of this technique is its resistance to key theft caused by inadequate memory scrubbing. Because the key used for decryption is different from the key used for encryption and the box performing the encryption does not have and cannot determine the corresponding decryption key, no harm is done to data security when the public key becomes compromised unless the key is also used for authentication.

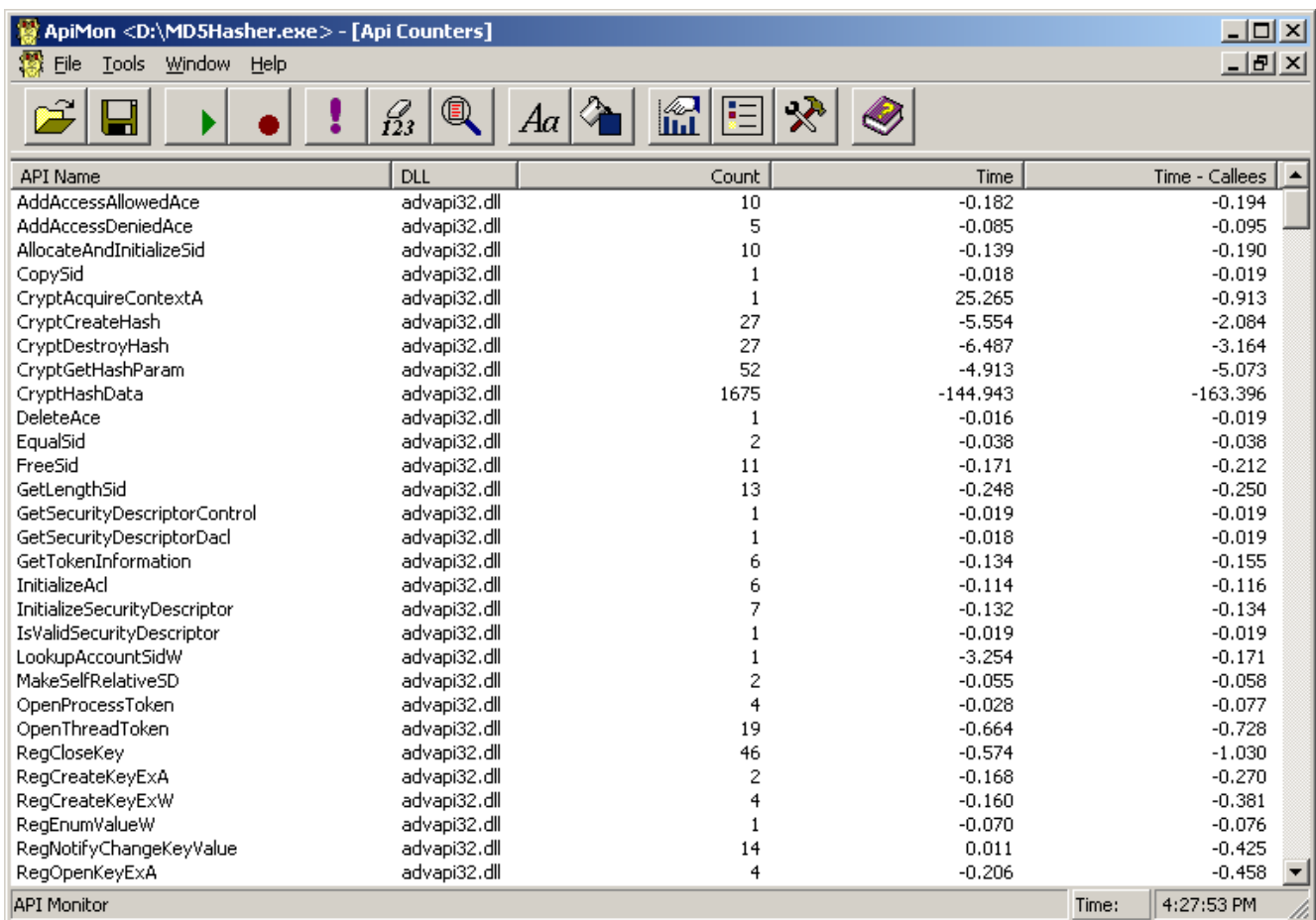
Analyzing Win32 API Dependencies

The Windows SDK provides tools that help analyze compiled code. Future releases of these SDK tools may include more code security analysis abilities as new techniques for spotting security problems in compiled code are developed. The simplest analysis is a complete log of API function calls made by the compiled code. Whatever else the code does of its own accord, you can be sure that it at least does whatever its Win32 API function calls facilitate. This sort of API call profile log can be generated either during program execution or from an analysis of the compiled code bytes with a tool that detects which import libraries the code is linked with. Dynamic DLL function calls by function vtable ordinal number after a LoadLibrary call can also be detected without resorting to actual execution of compiled code bytes. The following utilities provided with the Windows SDK require code to execute in order to log the API calls made but similar tools available from third parties do not.

Figure 10-9 shows the API Monitor program designed to intercept, log, and tally every call to every Win32 API function. By choosing to enable trace mode in the program's settings you are also given a complete trace log of every call intercepted by API Monitor in sequence. Even if you've written many Win32 applications in the past you probably don't have encyclopedic knowledge of every API function in memory so spotting calls that may be especially vulnerable to security flaws takes some effort but it's never a waste of time to profile and trace compiled code in a test bed before you allow it to execute on a production box. By default the trace log file is written to `apitrace.log` in the `%SYSTEMROOT%` directory. Typical API trace output is below:

LastError ReturnVal Name

```
0x00000000 0x0000001d GetModuleFileNameW 0x79170000 D:\WINNT\S 0x00000104
0x00000000 0x0000001d GetModuleFileNameW 0x79170000 D:\WINNT\S 0x00000104
0x00000002 0xffffffff GetFileAttributesW D:\WINNT\S
0x000000cb 0x00000000 GetEnvironmentVariableW COMPlus_In NULL 0x00000000
0x000000cb 0x00134c20 LocalAlloc 0x00000000 0x0000008a
0x000000cb 0x00000000 GetEnvironmentVariableW COMPlus_Co NULL 0x00000000
0x000000cb 0x00000000 GetEnvironmentVariableW COMPlus_Bu NULL 0x00000000
0x000000cb 0x00000000 GetEnvironmentVariableW COMPlus_Ve NULL 0x00000000
0x000000cb 0x00400000 GetModuleHandleW NULL
0x000000cb 0x00134218 LocalAlloc 0x00000000 0x00000014
0x000000cb 0x00000010 GetModuleFileNameW 0x00000000 D:\MD5Hash 0x00000104
0x000000cb 0x001323f8 LocalAlloc 0x00000000 0x00000010
0x00000002 HANDLE0000 CreateFileW D:\MD5Hash 0x80000000 0x00000001 0x00000000
```



The screenshot shows the API Monitor application window titled "ApiMon <D:\MD5Hasher.exe> - [Api Counters]". The window has a menu bar with "File", "Tools", "Window", and "Help". Below the menu bar is a toolbar with various icons. The main area displays a table with the following columns: "API Name", "DLL", "Count", "Time", and "Time - Callees". The table lists various API calls and their associated DLLs, counts, and times. The "API Name" column is sorted in descending order of "Count".

API Name	DLL	Count	Time	Time - Callees
AddAccessAllowedAce	advapi32.dll	10	-0.182	-0.194
AddAccessDeniedAce	advapi32.dll	5	-0.085	-0.095
AllocateAndInitializeSid	advapi32.dll	10	-0.139	-0.190
CopySid	advapi32.dll	1	-0.018	-0.019
CryptAcquireContextA	advapi32.dll	1	25.265	-0.913
CryptCreateHash	advapi32.dll	27	-5.554	-2.084
CryptDestroyHash	advapi32.dll	27	-6.487	-3.164
CryptGetHashParam	advapi32.dll	52	-4.913	-5.073
CryptHashData	advapi32.dll	1675	-144.943	-163.396
DeleteAce	advapi32.dll	1	-0.016	-0.019
EqualSid	advapi32.dll	2	-0.038	-0.038
FreeSid	advapi32.dll	11	-0.171	-0.212
GetLengthSid	advapi32.dll	13	-0.248	-0.250
GetSecurityDescriptorControl	advapi32.dll	1	-0.019	-0.019
GetSecurityDescriptorDacl	advapi32.dll	1	-0.018	-0.019
GetTokenInformation	advapi32.dll	6	-0.134	-0.155
InitializeAcl	advapi32.dll	6	-0.114	-0.116
InitializeSecurityDescriptor	advapi32.dll	7	-0.132	-0.134
IsValidSecurityDescriptor	advapi32.dll	1	-0.019	-0.019
LookupAccountSidW	advapi32.dll	1	-3.254	-0.171
MakeSelfRelativeSD	advapi32.dll	2	-0.055	-0.058
OpenProcessToken	advapi32.dll	4	-0.028	-0.077
OpenThreadToken	advapi32.dll	19	-0.664	-0.728
RegCloseKey	advapi32.dll	46	-0.574	-1.030
RegCreateKeyExA	advapi32.dll	2	-0.168	-0.270
RegCreateKeyExW	advapi32.dll	4	-0.160	-0.381
RegEnumValueW	advapi32.dll	1	-0.070	-0.076
RegNotifyChangeKeyValue	advapi32.dll	14	0.011	-0.425
RegOpenKeyExA	advapi32.dll	4	-0.206	-0.458

API Monitor Time: 4:27:53 PM

Figure 10-9: Use APIMon from the Windows SDK to Profile Executable Code

COM is essentially the Win32 object model. It's so prevalent in Windows OS code and applications built by Microsoft that it's now a de facto part of the Win32 API rather than an optional object oriented enhancement. The same thing is happening now with .NET managed code. With COM+ 1.5 and .NET managed code is being pushed as the platform for secure application development to replace classic Win32 API programming it's almost as if the two worlds have merged into an inseparable fusion; one part legacy

code with fundamental security problems and one part new code that is manageable and securable. Underlying every .NET managed application is a translation engine that takes Microsoft Intermediate Language (MSIL) byte code and converts it to legacy Win32 API calls.

While COM+ 1.5 and .NET Common Language Runtime foundations support secure code to a degree not possible through the pure Win32 API, these security enhancements do not yet go straight to the core of the OS. For the foreseeable future, the Windows platform will continue to expose a raw API layer that can be secured only in terms of blunt allow or deny rules based on the calling security context. Extremely important security improvements provided by .NET managed code such as evidence-based security and runtime call stack analysis don't protect calls directly to Win32 APIs.

Forensic profiles of executable code are essential to the practice of safe computing. Some executable code originates from your own programming work or the work of your colleagues and you can therefore conduct source code security reviews and select compiler security optimizations such as placing a canary on the stack with the /GS option in Visual C++ 7.0 in order to prevent buffer overflow exploits. Much of the code that your IIS box executes, however, comes from third party independent software vendors or from Microsoft. Since it is this code that is most likely to be targeted by Trojans and other attacks that exploit common vulnerabilities in code that has a wide install base, security flaws in your own custom code are less critical to protect against than flaws in other people's compiled code. The first and most important security assurance measure that must be implemented to protect against the execution of untrusted programs is an application initialization countermand layer. Next it's important to ensure that your software vendor has done everything possible to wrap their compiled code in safety assurance countermeasures. Finally, profiling Win32 API function calls and reviewing as much information as possible about models of vulnerable code gives you as much insight into the design implemented by a vendor. A practical security review always falls short of complete reverse engineering due to time constraints, but with the right tools and a test bed where debuggers and debug/checked builds of vendor code are available to help with this security analysis you can usually decide whether or not to trust particular compiled code in your production systems in a short amount of time.

It has long been an inside joke with assembly language programmers that there is value in the NOP instruction (No OPeration) for programmers who get paid by the byte since it allows them to increase the size of program code and thereby garner higher pay on a project. But NOP may have infosec value as well in that it can, if applied liberally throughout a program, force useful machine code instructions to move beyond the addressing range of branch instructions injected by an attacker and thereby complicate the search for a workable exploit based on the presence of some other vulnerability such as an unchecked buffer. There's also no reason for machine code to be ordered structurally in memory in an externally predictable fashion. So long as function pointers and function entrypoints are adjusted at runtime to reflect the actual virtual addresses that code must reference in order to invoke subroutines, machine code instructions that cause meaningful data processing can in principle be shuffled randomly in and around piles of NOP and buckets of null. Such techniques might force a program to execute top-down in its entirety, removing any chance of code fragments being misused or functions being called at a vulnerable midpoint address within the function body rather than at the

function's regular entry point. One thing's for certain: code safety assurance is an area of information security, forensics, and reverse engineering that is wide open for new innovation and improvement.

Chapter 11: ISAPI Health and Hardening

Many security flaws in IIS versions 4 and 5 came from preconfigured extensions and filters that conform to the Internet Server Application Programming Interface (ISAPI) DLL architecture. In fact, if you used either of these versions of IIS and removed every ISAPI filter or extension then your box resisted every worm that relied on IIS as a replication vector. It wasn't flaws in inetinfo.exe, w3svc.dll, or even the Active Server Pages script engine asp.dll that made IIS worms possible, they were made possible by flaws in Index Server (idq.dll) and other ISAPI extensions and filters installed by default or commonly activated by choice in IIS deployments. Because ISAPI is a compiled code extensibility programming feature of IIS, it is particularly vulnerable to common security problems like those described in Chapter 10.

Prior to IIS 6 it was possible for a developer or administrator to deploy ISAPIs in-process as part of inetinfo.exe and thereby give an ISAPI control over the SYSTEM security context. This feature was made part of IIS in spite of the decades-old infosec practice of running potentially-vulnerable network services under an unprivileged user account. More than any other single design decision, the in-process ISAPI option resulted in widespread system compromise when it was shipped to unsuspecting customers who neither understood the security risk it posed for production deployments nor realized that it made IIS vulnerable to remote exploit even in cases where the customer never used IIS or any of its preconfigured ISAPIs. Like much of the software produced by the industry even today, previous versions of IIS have default features that must be disabled before the software can be used safely.

Fundamentals of Reliable Extensions or Filters

To achieve success with any information system requires a minimum level of reliability, stability, and integrity sometimes referred to collectively as security. This type of security is a part of the definition of infosec, and it's never a waste of time to include it in a security analysis or secure computing initiative but there isn't always much you can do about fundamental design flaws in production systems except cross your fingers and wait for the upgrade. In some ways ISAPI is the Achilles heel in the anatomy of IIS. If it's damaged, the simple range of ability normally taken for granted becomes impossible (request processing stops). The Microsoft programmers responsible for inetinfo.exe could nail their respective secure coding tasks with pure infosec perfection and some junior programmer or intern writing a sample program could end up giving away customer boxes to any remote attacker who wants to own them simply because the sample includes an ISAPI that is present by default when the customer installs IIS. This isn't far from exactly what happened in the painful history of IIS. With IIS version 6 ISAPI is still a vulnerability for any Web application that uses either an extension or filter, but the vulnerability is contained and manageable.

System integrity includes its necessarily-resilient character with respect to failure isolation. It is never acceptable for faulty code in one application to bring down an entire server or impact processing of other applications. There were many ways in which ISAPI failures under IIS 4 and 5 would do both, violating this rule of failure isolation even when

applications were configured to run out-of-process. Most if not all of these known system integrity problems caused by the design of ISAPI and the inappropriateness of that design for every conceivable hosting scenario have been remedied in IIS 6. The boundaries that should have existed in the first place between privileged code running in kernel mode inside inetinfo.exe and application code running exclusively outside of inetinfo.exe are finally solid and reliable.

Several ISAPI features introduced with IIS 6 increase the ability of developers and administrators alike to build and manage hardened applications. As part of IIS 6 worker process isolation mode, application pools allocate resources and provide configuration settings for security, health, and performance of each Web application. Within each application pool the settings shown in Figure 11-1 control Idle Timeout, http.sys Request queue limit, CPU monitoring and number of worker processes to spawn if more than one is desired per garden.

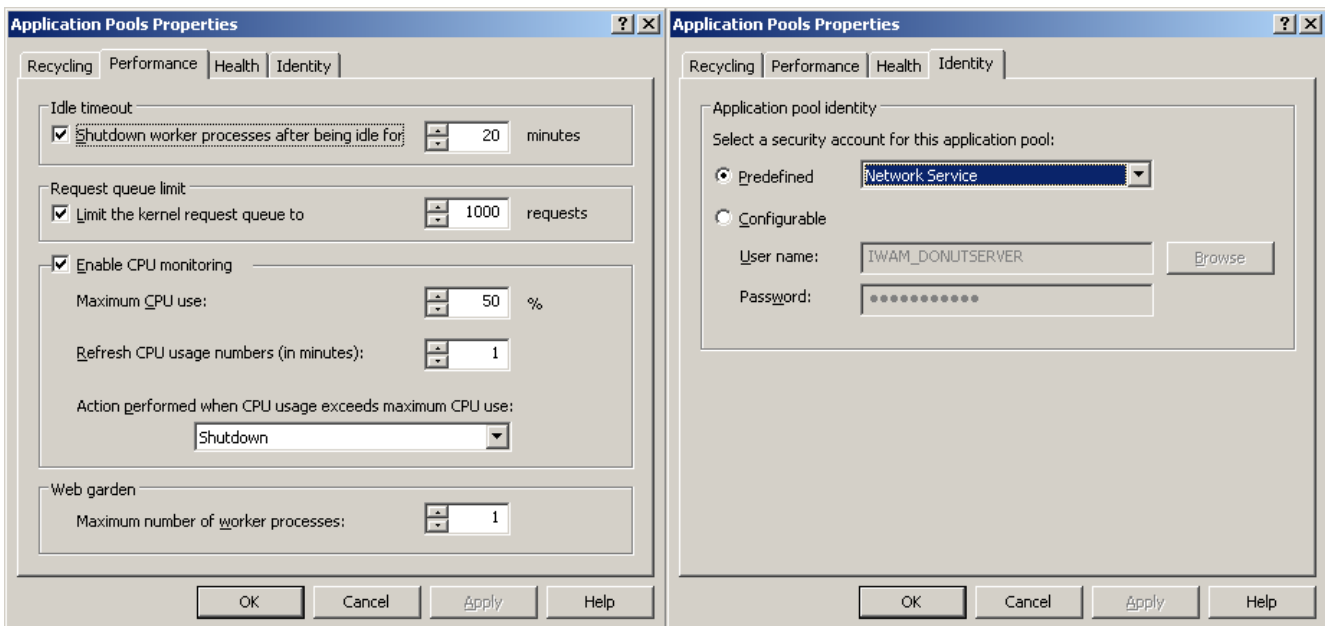


Figure 11-1: Performance and Identity Settings for IIS 6 Application Pools

Both ISAPI filters and extensions load only in worker processes under IIS 6; they no longer support activation within inetinfo.exe where vulnerabilities could result in the type of catastrophic security failures previously seen with ISAPI DLLs. By forcing ISAPI modules into worker processes rather than loading them into the inetinfo.exe process, IIS 6 prevents vulnerable ISAPI code from giving away remote control of a privileged security context through malicious hijacking of a process that owns the SYSTEM security token. The exception to this rule (there always has to be at least one, right?) is IIS 5 Isolation Mode, a new setting that allows IIS 6 to revert from its worker process isolation mode to the legacy model where ISAPI filters will load in inetinfo.exe. Filters that register for the SF_NOTIFY_READ_RAW_DATA event must use IIS 5 Isolation Mode because these raw filters can't be configured at the individual Web site level they only work as part of the Web service.

Malformed Requests with Malicious Payloads

Intentionally malformed requests sent to vulnerable ISAPIs are the basis of most automated attacks on IIS. One reason such malformed requests are able to exploit bugs in ISAPI code is that programmers access content of the HTTP headers by way of a `GetServerVariable` callback function rather than parsing these values from raw request data. This makes the programmer lazy and predisposes them to the presumption that IIS will never give them anything harmful to security of the ISAPI. However, the `EXTENSION_CONTROL_BLOCK` (ECB) structure passed in to `HttpExtensionProc` entry-point function includes the character buffers of arbitrary length listed in Table 11-1. These buffers are filled from the HTTP request and are therefore under the complete control of the remote sender.

Table 11-1: ECB Character Buffers of Remote Origin

ECB Character Buffer	ServerVariables Equivalent
<code>lpszMethod</code>	<code>REQUEST_METHOD</code>
<code>lpszQueryString</code>	<code>QUERY_STRING</code>
<code>lpszPathInfo</code>	<code>PATH_INFO</code>
<code>lpszPathTranslated</code>	<code>PATH_TRANSLATED</code>
<code>lpszContentType</code>	Content type of client data

An ISAPI developer should never use the values supplied in the ECB. There is no need to do so, for one thing, because each value can be obtained through a call to the `GetServerVariable` callback function instead. Programmers tend to use the values listed in Table 11-1 out of the ECB instead because it's easier. But to do so one must explicitly ensure that the length does not exceed the size of buffers allocated to hold copies of these values. The call to `GetServerVariable` verifies the output buffer size supplied by reference, whereas the ECB character buffers are of undetermined length and require the programmer to perform a buffer copy while making the assumption that there is proper null-termination of that buffer marking its end point.

Careful ISAPI Development and Deployment

As a developer looking for a third-party ISAPI to solve a particular problem or contemplating rolling your own, consider ISAPI your last resort if there is no other way to achieve the performance and scalability you require. There is nothing inherently wrong with ISAPI, from an infosec perspective, it is just as risky as any other compiled code in any network service that is exposed to and must process bytes of unknown origin. As long as an ISAPI does not expose a remote exploitable buffer overflow vulnerability, there are few other security concerns inherent in the ISAPI architecture that you have control over as a developer. There are, of course, any number of ways to write bad code. And there are security-related `ServerVariables` and `ServerSupportFunctions` that many ISAPIs will want to use and should make use of properly to avoid malfunction. But the bulk of security assurance for ISAPIs involves the same code hardening procedures as one would normally apply to any compiled code meant to be used within a DLL loaded into multiple processes in a multithreaded services environment. If you aren't willing or able to follow complex and often subtle secure coding best practices for writing thread-safe DLL code for this type of environment, then you have no business writing ISAPIs.

In IIS 4 and 5, `GetExtensionVersion` is called from within `inetinfo.exe` under the System security context. Whereas `DllMain` is called in the impersonation account used for request

processing that invokes the ISAPI extension. One exception to this rule is the case where inetinfo.exe process shutdown or garbage collection cause the ISAPI to unload. In this case, as with garbage collection of in-process COM objects, the DLL_PROCESS_DETACH finalization call intoDllMain may also occur in the System context.

As an administrator you must demand access to a complete source code security review of each ISAPI, and you should inquire as to the qualifications of any person who writes ISAPI code that you plan to deploy on your boxes. There are almost always alternatives to ISAPI that would work well enough, although not as efficiently, and offer lower inherent risk. IIS 6 reworks the ISAPI extension model around worker process isolation mode and in so doing makes it easier for experienced developers to achieve increased fault tolerance and DoS resistance in ISAPI code. The ability to replace relatively hard-to-write ISAPI filters with a new type of ISAPI extension called a global interceptor also helps to relocate sensitive code to make it more manageable and resilient under heavy load. Rethinking ISAPI in security terms and carefully reviewing the architecture of applications that make use of ISAPIs may provide the most significant return on security investment available under IIS 6.

The Microsoft Foundation Class (MFC) Library version 7.0 provided with Visual C++ .NET includes four classes that help to build ISAPIs. By using the security optimizations available in this version of Visual C++ (such as /GS guard stack mode described in the previous chapter) along with MFC, you can more easily create trustworthy compiled code for ISAPI filters and extensions. MFC classes CHttpServer and its request context class CHttpServerContext are the basis of an ISAPI extension, while ISAPI filters use classes CHttpFilter and CHttpFilterContext. The methods inherited from either of these classes can expose your ISAPI to raw input from the client request and all its attendant risk and responsibility. There are also a few known security issues with MFC ISAPIs and simple ways to avoid problems as a result of these issues.

MFC Parse Maps

When a request is received by IIS for an ISAPI extension built with MFC, the incoming request is processed by a series of parsing macros in an ISAPI parse map. The purpose of these macros is to dispatch the request to the right processing code within the ISAPI. The parse map macros are subject to buffer overflow vulnerabilities because they create and fill buffers rather than simply examining the ones provided by IIS in the request context provided when the ISAPI is triggered. The parse map macros consist of the following mapping instructions within BEGIN_PARSE_MAP and END_PARSE_MAP delimiters.

```
ON_PARSE_COMMAND  
ON_PARSE_COMMAND_PARAMS  
DEFAULT_PARSE_COMMAND
```

The DEFAULT_PARSE_COMMAND maps request handling to the designated method on the CHttpServer-derived object that is invoked if no explicit command is found in the query string of the request. The MFC parse map determines whether a command is present by looking for name/value pairs in the URL's query string.

ON_PARSE_COMMAND and ON_PARSE_COMMAND_PARAMS are optional but must be present in pairs when they are used in parse map macros to extract name/value pairs from the incoming request, store them in temporary buffers, and pass pointers to these buffers in calls to appropriate methods. A race condition was discovered in MFC parse map macro implementation as described in Q260172:

Knowledge Base Article Q260172 FIX: MFC ISAPI Parse Functions Fail Under Stress on Multiple-CPU Computers

A classic stack buffer overflow vulnerability also existed in the MFC ISAPI CHttpServer::OnParseError base method shipped with Visual C++ 6.0 prior to Service Pack 3. Knowledge base article Q216562 details the fact that originally this MFC ISAPI code used a fixed-length 256-character buffer to receive bytes of arbitrary length provided in the request sent to the server. An intentionally malformed request that triggered the MFC parse map error handler could stuff malicious bytes into this fixed-length character buffer, forcing an overflow condition and potentially taking control of the process. Two additional knowledge base articles are worthy of note because they illustrate the extent to which the MFC parse map macros depend on a fixed interpretation of request command structure. There may be additional security problems not yet identified in MFC parse map macro code, and therefore special care should be taken to review the safety of your MFC ISAPIs.

Q169109 PRB: Parse Maps Do Not Handle Multi-select List Boxes

Q174831 Using Check Boxes and Radio Buttons with MFC Parse Maps

ON_PARSE_COMMAND relies on a set of predefined constant arguments to determine the type of parameters that are expected as part of a particular command. The values are ITS_EMPTY, ITS_ARGLIST, ITS_RAW, ITS_PSTR, ITS_I2, ITS_I4, ITS_R4, ITS_R8, and ITS_I8. Each of these arguments, of which there must be at least one and the first three, if used, must be used alone, translates to a certain parameter type. ITS_PSTR represents a null-terminated string pointer, ITS_RAW represents a void * along with a DWORD value indicating the number of bytes in the buffer pointed to by the void * parameter, and ITS_ARGLIST is a pointer to an instance of CHttpArgList. The remaining argument constants indicate numeric type parameters passed by value. When you use ON_PARSE_COMMAND as part of an MFC parse map your ISAPI is dependent on the safety of the macro code after it expands and compiles. The race condition found and fixed as described in Q260172 and the buffer overflow condition found and fixed as described in Q216562 may only have been applicable in the case of ITS_PSTR parameters, but this isn't surprising since it is the buffer handling that is most vulnerable to begin with not simple fixed-length numeric parameter passing by value. To err on the side of caution you can avoid using the MFC parse map macros for anything other than default method mapping or ITS_RAW. Either of these command types should be as safe as safe gets with an MFC ISAPI's initial request dispatching interface. Allowing MFC to do request parsing for you saves some coding but also creates security issues that are difficult to quantify even after reading isapi.cpp and the other MFC source files.

Custom ISAPIs Without MFC

Knowledge base article Q216562 is extremely important for you to review it if you plan to use CHttpServer as the basis of an ISAPI extension. The buffer overflow vulnerability in CHttpServer::OnParseError represents the worst-case scenario for MFC ISAPI; simply using CHttpServer as the basis of an ISAPI is sufficient to render the ISAPI vulnerable to remote exploit unless you follow the instructions in Q216562. There may be other vulnerabilities in MFC ISAPIs yet to be discovered. This is an issue that you must keep close tabs on in order to ensure the safety of your MFC ISAPIs.

FIX: Access Violation in MFC ISAPI with Large Query String
PSS ID Number: Q216562

The information in this article applies to:

Microsoft Visual C++, 32-bit Enterprise Edition 5.0, 5.0sp1, 5.0sp2, 5.0sp3, 6.0

Microsoft Visual C++, 32-bit Professional Edition 5.0, 5.0sp1, 5.0sp2, 5.0sp3, 6.0

Microsoft Visual C++, 32-bit Learning Edition 6.0

The safest way to code ISAPIs is without the help of MFC. At least without using CHttpServer and its request context class CHttpServerContext as the basis of an ISAPI extension and without using classes CHttpFilter and CHttpFilterContext as the basis of a filter. The ISAPI entry-point functions are simple to expose from a regular DLL, and there's reason to believe that adding 2,448 lines of code from isapi.cpp, 688 lines of code from afxisapi.h, 140 lines of code from afxisapi.inl, compiling other MFC #includes and then linking with the MFC libraries all this code depends on will not make your ISAPI more secure. You only really need the following lines of code to create a functional ISAPI extension without using MFC.

```
#include <httpext.h>
static HINSTANCE g_hInstance = NULL;
HINSTANCE __stdcall AfxGetResourceHandle()
{ return g_hInstance;
}
BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason,
LPVOID lpReserved) {
if(ulReason == DLL_PROCESS_ATTACH) {
g_hInstance = hInst; }
return TRUE;
}
extern "C" DWORD WINAPI HttpExtensionProc(EXTENSION_CONTROL_BLOCK *pECB) {
DWORD dwRet = HSE_STATUS_SUCCESS;
UINT_PTR uip = sizeof(DWORD);
if(!IsBadReadPtr(pECB,uip)) {
if(IsBadReadPtr(pECB,pECB->cbSize)) {
dwRet = HSE_STATUS_ERROR; }}
return dwRet;
}
extern "C" BOOL WINAPI GetExtensionVersion(
HSE_VERSION_INFO *pVer) {
if(!IsBadWritePtr(pVer,sizeof(HSE_VERSION_INFO))) {
pVer->dwExtensionVersion = HSE_VERSION;
}
```

```

pVer->lpszExtensionDesc[0] = '\0';
return TRUE; }
else {
return FALSE; }
}
extern "C" BOOL WINAPI TerminateExtension(DWORD dwFlags) {
return TRUE;
}

```

The code shown here is a viable starting point for an ISAPI extension. It is simple and conservatively makes explicit memory read and write permission checks prior to attempting to use memory blocks passed in from IIS. The kernel32 exports `IsBadReadPtr` and `IsBadWritePtr` provide this runtime safety check. The complexity of writing provably-safe ISAPI filters from scratch is a little higher, but as with the ISAPI extension entry-point functions, the ISAPI filter entry-point functions `GetFilterVersion`, `HttpFilterProc`, and `TerminateFilter` are not difficult to write or copy and paste from sample code or the `isapi.cpp` MFC source file. You can be certain of the quality and safety of ISAPI code only if you have access to all of it, you understand every line, and you personally compile and link the code yourself or you know you can trust the person who did.

Global Interceptors

In IIS 4 and 5 ISAPI filters load into `inetinfo.exe` even when all WAM applications are configured out-of-process making ISAPI filters high-risk points of failure in the privileged `inetinfo.exe` process. Remote exploits of which can allow execution of arbitrary code or alter the request processing behavior of IIS to cause arbitrary files or other data to be returned in a response. In IIS 4 and 5, Web Application Manager (WAM) places ISAPI filters in memory and chains calls to them before and after calls to the ISAPI extension, CGI executable program, or default static file delivery code contained in `w3svc.dll`. Any WAM application marked out of process performs its application-specific processing, or its static content delivery, in a different process from the one in which its ISAPI filters reside. IIS 6 resolves this security problem by forcing all ISAPI filters into the same least privileged worker process that provides the rest of the application-specific request processing where the least privileges possible, the minimum required for work to be done, are granted to the security context in which the worker process executes. In addition, due to the superior manageability of ISAPI extensions versus filters, a new type of extension has been created called a global interceptor with a wildcard application mapping. Figure 11-2 shows how this wildcard mapping is established for a global interceptor ISAPI.

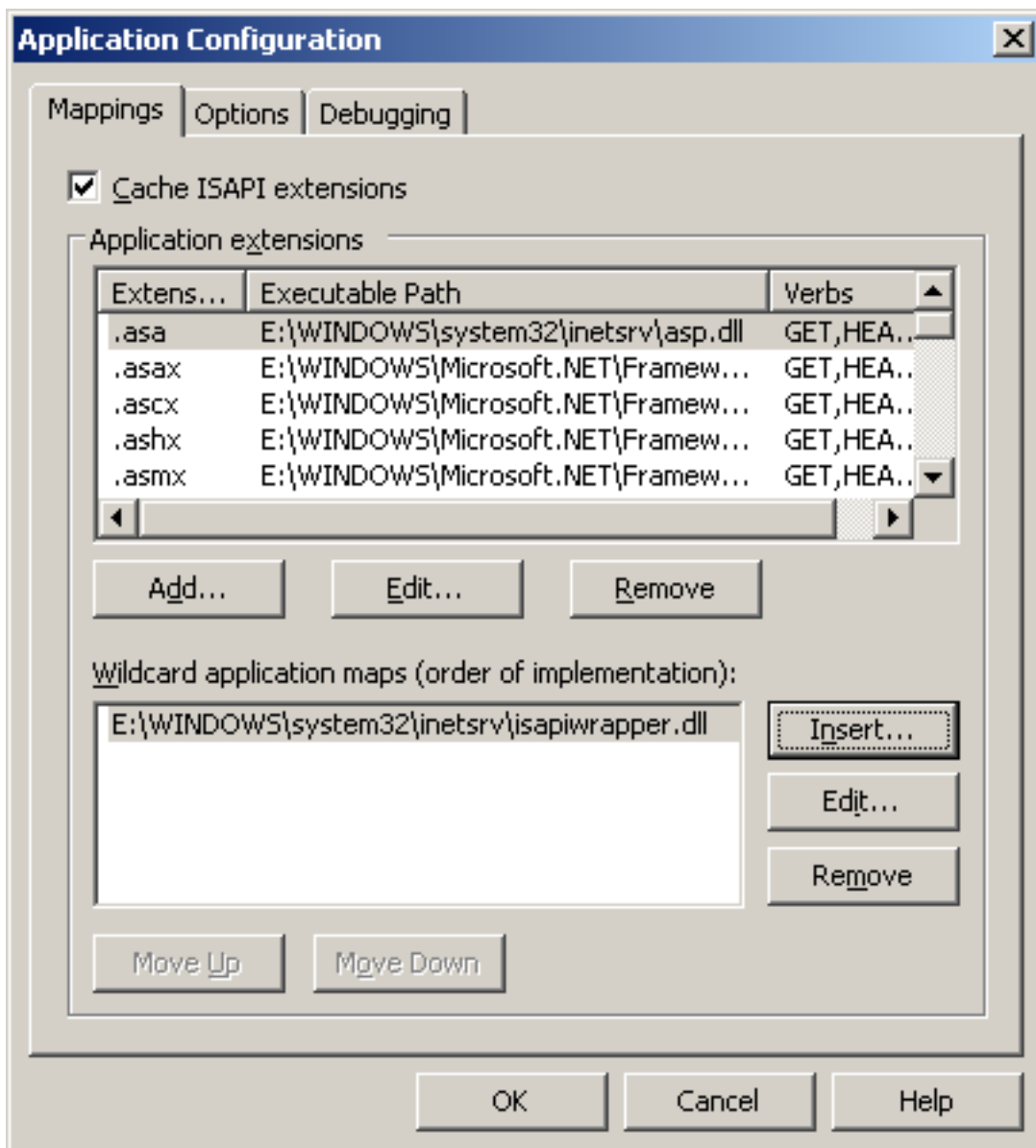


Figure 11-2: Global Interceptor ISAPIs Support Wildcard Application Mappings

Request Chaining and Rewriting

One of the most valuable additions to ISAPI made with IIS version 6 is the ability to chain together request processing by accepting the client request and chaining it together with the output produced by forwarding the request on to another URL. This is the basis of any reverse proxy, and it also has layered security potential where requests can be sanitized, scrubbed, validated and rewritten prior to being passed on to another server or a different URL referencing the same server that can complete processing of the request. A new HSE_REQ_EXEC_URL ServerSupportFunction is provided in IIS 6 that enables this type of extended local or remote processing. When coupled with global interceptor wildcard script mapping, the ability to easily invoke the processing of another URL from within the context of an ISAPI is sure to find a variety of important uses and security applications.

As a tool for secure modular ISAPI design, request chaining and rewriting exceeds the normal server-side include mechanism in usefulness. When processing a conventional server-side include directive, the include target must be accessible to the server box by way of the filesystem and its contents must be compatible with the script engine currently processing the request. The item so included sees the raw client request and is therefore vulnerable to its potentially-malicious payload. HSE_REQ_EXEC_URL makes it possible for an ISAPI to pull together processing resources from disparate locations, platforms, programming languages, and insecure legacy systems that need more than just firewalls and intrusion detection systems to protect them from attack. Only available from within ISAPI version 6 extensions, this new ServerSupportFunction is complemented by an older programming technique whereby ISAPIs can layer-in security wrappers and other layered processing code.

Layered Security Wrapping of ISAPIs

Third-party code safety assurance rests on accurate forensic analysis including your own review of binary code in advance of unleashing it on a production server. Without access to the source code and the time and knowledge necessary to review it completely, and a guidebook to help with your forensic security analysis, there's just no way to know for sure that everything an ISAPI can ever do is completely safe, harmless, or appropriate. You don't get this certainty with third-party ISAPIs yet running any ISAPI without this certainty is unacceptable risk. The solution to this dichotomy is a simple one: create a profile of specific requests that you know to be safe for each ISAPI. Prove that certain requests result only in appropriate behavior and then deny everything else. Preventing an ISAPI from receiving any request that is not provably safe for it to execute is the only defense against unknown functionality that might lie hidden in the internal repeated bifurcations branching from DIIMain that fork into uncontrollable complexity to produce an ISAPI's vast runtime potential.

The following ISAPI extension code implements just such an ISAPI wrapper that examines the request received from the client and enforces a simple pattern matching rule, that the QueryString must contain only "A=A" and the total size of the client request must not exceed 512 bytes. In addition, the length of QueryString is limited to 256 bytes, and this length limit is checked prior to any use of the buffer. You can use this ISAPI wrapper to protect any ISAPI extension from receiving requests that contain unnecessary or potentially dangerous data. With a little more code the wrapper can implement any input sanitizing algorithm you prefer, including character encoding validation, removal of extraneous HTTP headers, or other EXTENSION_CONTROL_BLOCK rewriting rule. Simply allocate heap memory for your replacement EXTENSION_CONTROL_BLOCK and pass it to the protected ISAPI instead. In the code as shown the protected ISAPI is ASP.DLL.

```
#include <httpext.h>
static HINSTANCE g_hInstance = NULL;
static CRITICAL_SECTION critical;
static HMODULE hMod = NULL;
static PFN_HTTPEXTENSIONPROC fp = NULL;
#define PROTECT "D:\\WINNT\\System32\\inet_srv\\asp.dll"
#define COMMAND "A=A" // QUERY_STRING lstrcmpi
```

```

#define BUFLLEN 512 // maximum request size
#define QSBUFLLEN 256 // maximum QueryString
HINSTANCE __stdcall AfxGetResourceHandle()
{ return g_hInstance; }
BOOL WINAPI DllMain(HINSTANCE hInst, ULONG ulReason,
LPVOID lpReserved) { if(ulReason == DLL_PROCESS_ATTACH) {
    InitializeCriticalSection(&critical);
    g_hInstance = hInst; }
else if(ulReason == DLL_PROCESS_DETACH) {
    DeleteCriticalSection(&critical); }
return TRUE; }
extern "C" DWORD WINAPI
HttpExtensionProc(EXTENSION_CONTROL_BLOCK *pECB) {
HSE_VERSION_INFO vinfo;
PFN_GETEXTENSIONVERSION fpgettext = NULL;
char *errmsg = "Access Denied.";
DWORD dwerrmsg = strlen(errmsg);
DWORD dwRet = HSE_STATUS_SUCCESS;
UINT_PTR uip = sizeof(DWORD);
BOOL bError = true;
char buf[BUFLLEN];
char qsbuf[QSBUFLLEN];
DWORD dwbuflen = BUFLLEN;
if(!IsBadReadPtr(pECB,uip)) {
    if(!IsBadReadPtr(pECB,pECB->cbSize)) {
        if(pECB->cbTotalBytes <= BUFLLEN || pECB->cbAvailable
<= pECB->cbTotalBytes) {
            if(pECB->GetServerVariable(pECB->ConnID,
"ALL_RAW",buf,&dwbuflen)) {
                dwbuflen = QSBUFLLEN;
                if(pECB->GetServerVariable(pECB->ConnID,
"QUERY_STRING",qsbuf,&dwbuflen)) {
                    if(lstrcmpi("A=A",buf) == 0) {
                        EnterCriticalSection(&critical);
                        if(hMod == NULL) { hMod = LoadLibrary(PROTECT);
                        if(hMod != NULL) {
                            fpgettext = (PFN_GETEXTENSIONVERSION)GetProcAddress(
hMod,"GetExtensionVersion");
                            if(fpgettext != NULL) { if(fpgettext(&vinfo)) {
                                fp = (PFN_HTTPEXTENSIONPROC)GetProcAddress(
hMod,"HttpExtensionProc"); }}}}
                            LeaveCriticalSection(&critical);
                            if(hMod != NULL && fp != NULL) { dwRet = fp(pECB);
                                bError = false; }}}}
                    if(bError) {
                        pECB->WriteClient(pECB->ConnID,errmsg,
&dwerrmsg,HSE_IO_SYNC); }
                    return dwRet; }
                }
            }
        }
    }
}
extern "C" BOOL WINAPI GetExtensionVersion(

```

```

HSE_VERSION_INFO *pVer) {
if(!IsBadWritePtr(pVer,sizeof(HSE_VERSION_INFO))) {
    pVer->dwExtensionVersion = HSE_VERSION;
    pVer->lpszExtensionDesc[0] = '\0';
    return TRUE; }
else { return FALSE; }}
extern "C" BOOL WINAPI TerminateExtension(DWORD dwFlags){
return TRUE; }

```

The ISAPI wrapper is first and foremost just an ISAPI extension. There is nothing new or special about its design. Building from the basic codebase shown previously as an example of a non-MFC ISAPI with a couple application integrity fundamentals included, the wrapper code uses fixed-sized stack buffers in calls to `GetServerVariable`. These calls return a Boolean value indicating success or failure, and failure is possible when the size of the fixed-length stack buffer passed by reference to the callback function as an output buffer is too small to hold the entire value requested. In this way the wrapper code is able to detect a violation of its rule restricting requests by size in bytes of the entire request and also of just its `QUERY_STRING` parameters. A critical section is used to manage the loading of the protected ISAPI, named in the `PROTECT #define`, and calls to `GetProcAddress` for the ISAPI entry-point functions it exports. When it is confirmed that the request meets the safety requirements imposed by the wrapper, including the presence of the explicit command string "A=A" in `QueryString`, the pointer to the protected ISAPI's `HttpExtensionProc` is used by way of pointer dereferencing to hand off processing of the request. Otherwise an error message is displayed to the client.

When this security-hardened ISAPI wrapper technique is combined with global interceptor wildcard mapping it's plain to see the potential to layer such an ISAPI extension, filter-like, into all request processing. By mapping your ISAPI wrapper DLL to any file type as the script engine for IIS to invoke when requests are received, you achieve something similar to an ISAPI filter but in the safer and easier-to-code context of an extension. Another benefit of extensions over filters is that they have access to enhanced health management features in IIS 6. It's possible to execute an ISAPI extension explicitly from ASP script or any file type that is mapped to the server-side include directive parser (`ssinc.dll`) ISAPI as well. More about this type of late-bound dynamic ISAPI chaining can be found later in this chapter.

Managed Healthcare for ISAPI Extensions

Worker process isolation mode in IIS 6 provides an optional ping of health that enables Web Administration Service (WAS) to verify periodically that a process pool still has threads available to do work. WAS resides inside `inetinfo.exe` along with `http.sys` and the other kernel-mode IIS code. The ping of health is a local procedure call made into an application pool by WAS that any thread in the process pool can handle. As long as the call succeeds, WAS knows that requests for the applications pooled for hosting within the worker process are still being serviced. However, WAS does not learn through this mechanism how many of the threads in the worker process pool have encountered deadlocks, infinite loops, or other failure conditions that take them out of service. A worker process that stops responding to the ping of health is recycled according to the recycling parameters established for the pool.

Worker Process Recycling

The ping of health, when enabled, applies to each worker process in a Web garden's application pool. This setting is turned on, and other settings for failure detection thresholds are set, by way of the Health tab in the Application Pools Properties sheet. Figure 11-3 shows both the Health and the Recycling tabs. In addition to these settings each ISAPI that executes within a worker process can now alert IIS that it has entered an unhealthy state that may require process recycling to correct. A new ServerSupportFunction HSE_REQ_REPORT_UNHEALTHY is used by an ISAPI to deliver this notification to IIS and to request recycling. When too many threads in a worker process report unhealthy, WAS recycles the process.

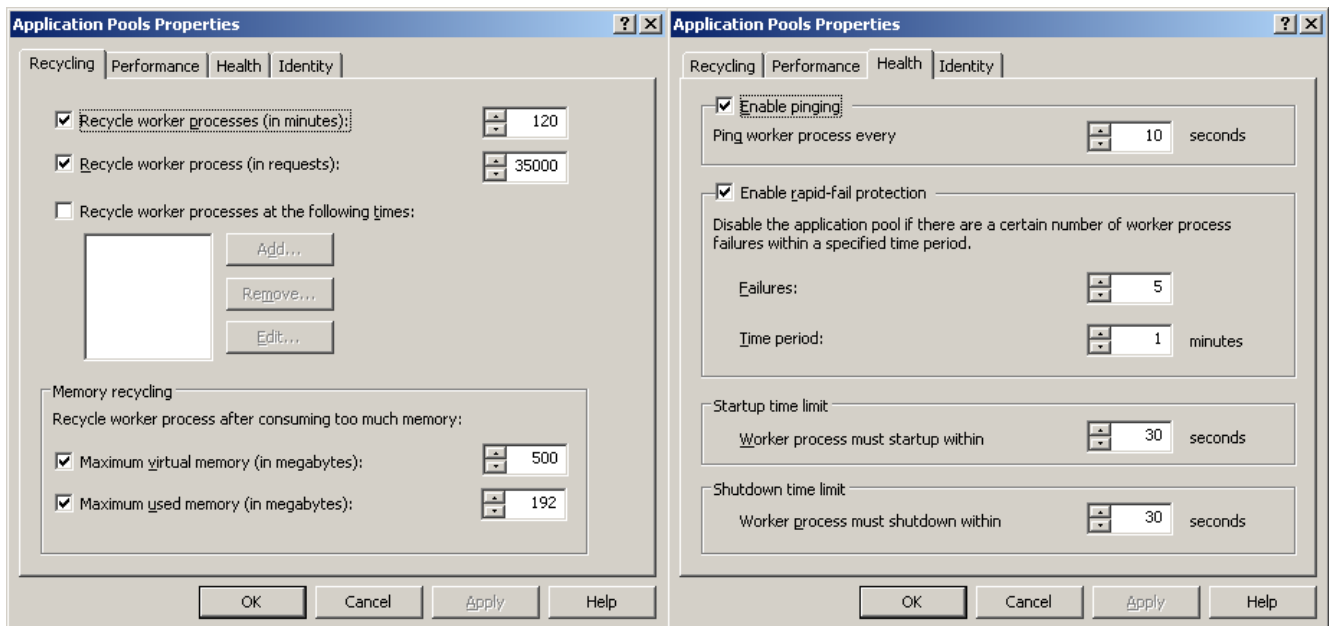


Figure 11-3: Recycling and Health Settings for Application Pools

WAS is also able to recycle a worker process based on its consumption of memory and CPU resources. Recycling can be scheduled automatically based on elapsed time (in minutes) since process launch, and number of requests processed. When WAS decides that process recycling is necessary, the faulty worker is taken out of service while it is allowed to complete whatever remaining processing it has left to do and is still capable of handling. A time limit is imposed on worker process shutdown as it is prepared for recycling. WAS creates a replacement process in the application pool and starts the worker process servicing requests before the old worker process stops. Because http.sys manages the IIS version 6.0 listening sockets in kernel-mode threads, each TCP connection accepted by http.sys is not dependent on the health and vitality of any worker process that may be recycled in the midst of processing. When a worker process is recycled, http.sys preserves connections with the HTTP client and automatically initiates a new worker process. From the perspective of the client there is no failure only a more lengthy time period between request and response.

Worker Process Kernel-Mode Overlapped I/O

Through a feature known as vector send IIS 6 enables a user-mode application to perform an optimized kernel-mode buffer write operation using overlapped I/O. The technique is a simple one and it has been around for some time but it hasn't been available until now for ISAPI applications to use in conjunction with IIS threads that send data over the network through Windows Sockets 2 API calls. With worker process isolation mode removing the option of in-process ISAPI for bandwidth- and performance-intensive Web applications in order to gain improved security and reliability, it would seem that every response written to the client would need to marshal data between the worker process and inetinfo.exe so that http.sys can manage the transmission of the response buffer. This means at least one buffer copy, from memory accessible only to the user-mode worker process to memory accessible to the inetinfo.exe process prior to each buffer send operation. Depending on whether the entire response is buffered or sent in chunks (possibly using HTTP 1.1 chunked transfer encoding) there may be many context switches and many wasteful buffer copies as data is marshaled from user-mode to kernel-mode inetinfo.exe process.

Enter the vector, a shared memory block allocated by user-mode code and passed as part of a vector array to kernel-mode http.sys so that it can, at its earliest opportunity, read directly from the shared memory vector rather than performing any buffer copies or multiple context switches. This technique is also known as scatter/gather overlapped I/O because an assortment of buffers (vectors) are filled and then scattered, tossed over to other code for its exclusive use, after which time the buffers are gathered up again for reuse. The code that fills the vector buffers receives notification when they are no longer required because their contents have been processed successfully. During the interim period the buffers are off-limits for writing or reuse; this is termed overlapped I/O because the buffer filled with bytes in the send operation is the same buffer used by the protocol stack to write bytes onto the wire. When performed in reverse, overlapped I/O can also be used to receive data using vectored buffer reads where bytes from the wire are stored by the protocol stack directly into the shared memory that an application uses to process this inbound data.

Vector Send Worker Process Kernel-Mode Overlapped I/O is exposed for use by your IIS 6 ISAPI extensions through a new HSE_REQ_VECTOR_SEND added ServerSupportFunction. In addition, kernel-mode http.sys cache is available for any vector send overlapped I/O-compatible ISAPI. An ISAPI that wishes to have its response cached by http.sys simply adds HSE_IO_FINAL_SEND and HSE_IO_CACHE_RESPONSE to dwFlags passed with the ServerSupportFunction HSE_REQ_VECTOR_SEND. More on this topic can be found in the IIS 6 SDK.

Browser Client Context ISAPI W3Who.dll

As part of auditing security before, during, and after deployment of any Web application built with IIS, one ISAPI in particular is valuable because it displays detailed information about the security context in which a request is processed. The ISAPI is called the Browser Client Context Tool (w3who.dll) and its diagnostic output is shown in Figure 11-4. The w3who.dll is provided as part of the Windows 2000 Server Resource Kit.

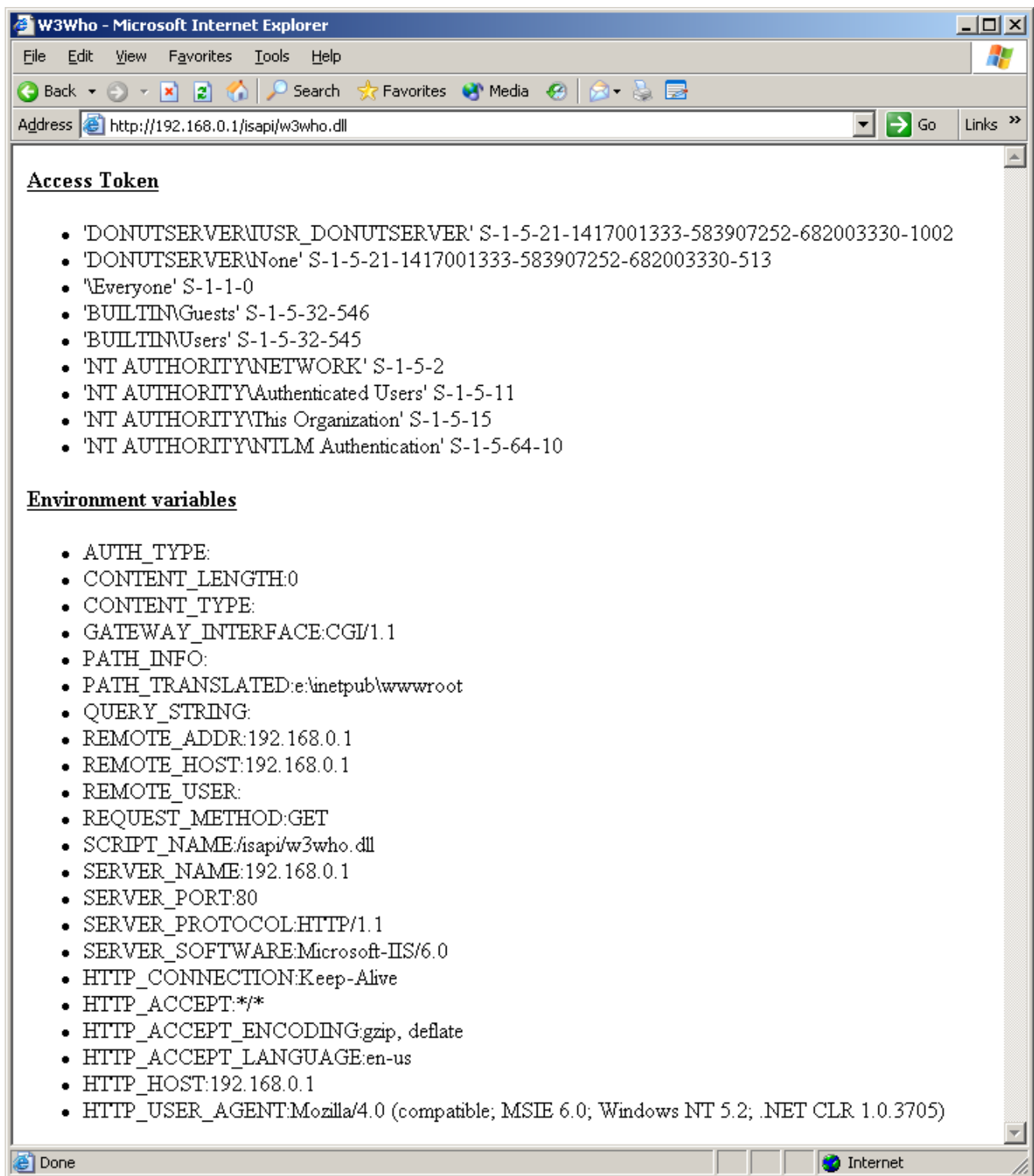


Figure 11-4: Diagnostic ISAPI W3Who Browser Output

See also Microsoft Knowledge Base Article Q318709 HOW TO: Use the Browser Client Context Tool (W3Who.dll) in Internet Information Services 5.0

To download the Browser Client Context Tool (W3Who.dll) visit www.microsoft.com/windows2000/techinfo/reskit/tools/existing/w3who-o.asp

To install w3who.dll requires the same steps as you would use to deploy any ISAPI extension, which under IIS 6 includes adding a new Web service extension using the MMC as depicted in Figure 11-5. Under IIS 6 an Application Mapping to an ISAPI extension or an ISAPI requested explicitly by a client is unavailable for use until a new Web service extension is created. Web Service Extensions are the equivalent of activation level security policy settings to allow or deny ISAPIs. This additional layer of protection against arbitrary deployment of ISAPIs to an IIS server provides a valuable administrative safeguard that didn't exist in previous IIS versions that allowed any ISAPI to activate on the server in any directory marked executable.

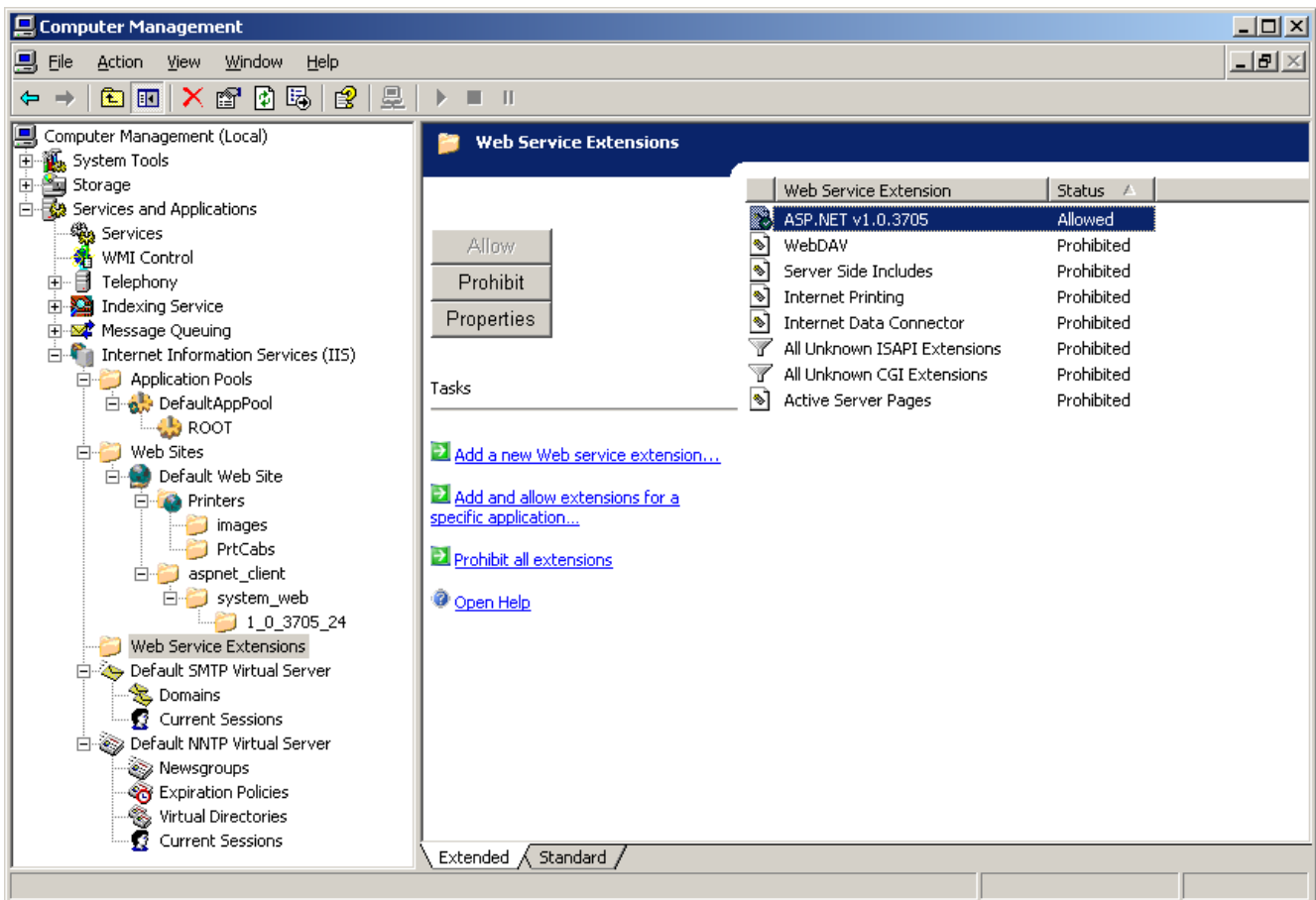


Figure 11-5: Web Service ISAPI Extensions are Prohibited by Default in IIS 6

To deploy and configure the Browser Client Context Tool under IIS 6 first download and install w3who.dll in a Web services directory that is marked as executable (granted "Scripts and Executables" execute permissions) in the metabase. Then open MMC and locate the Web Service Extensions settings under Internet Information Services (IIS) as shown in Figure 11-5. Click Add a new Web service extension... under Tasks. The window shown in Figure 11-6 appears, enabling any number of ISAPI extensions to be configured as Allowed for activation within the context of request processing. Once this step is complete, client requests directed to the w3who.dll ISAPI extension will cause it to activate and perform its function.

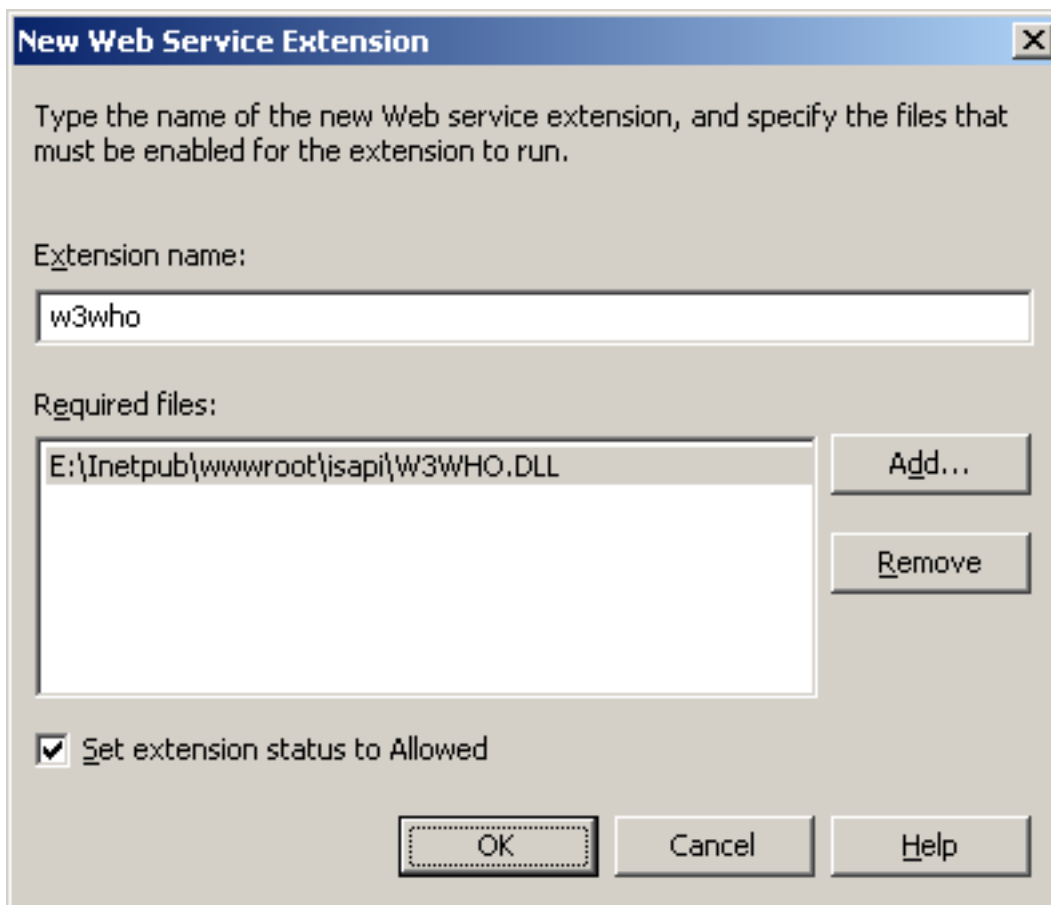


Figure 11-6: Add a Web Service Extension for w3who.dll in IIS 6

Common Gateway Interface ISAPI Alternatives

ISAPI in IIS 6.0 provides new features such as custom errors, ExecuteURL-style chaining with HSE_REQ_EXEC_URL to enable the new global interceptor type of ISAPI Extension providing request preprocessing ability previously possible only as part of an ISAPI Filter, significant performance improvement using scatter/gather overlapped IO with the VectorSend ServerSupport function to hand off buffers to http.sys for transmission to the client in a single user-to-kernel-mode transition, and health reporting using ServerSupport function HSE_REQ_REPORT_UNHEALTHY within worker process isolation mode where unhealthy workers can be recycled. To top it all off, version 6.0 provides Unicode ISAPI support not available previously so that URLs encoded using the UTF-8 character set can be parsed properly and values accessed through ServerVariables can contain UTF-8 encoded data. There are many benefits to ISAPIs and the only drawback is that programming them to be robust and secure is more difficult than the alternatives and requires skill with the C++ language.

A simpler, and perhaps more secure way to create server-side dynamic content that is active rather than static is to use the conventional Common Gateway Interface (CGI).

When deciding how to handle each HTTP request, IIS first parse the URL to locate an appropriate Application Mapping where file extension indicates which ISAPI extension or executable, such as a script engine, that IIS should invoke. If no Application Mapping is

configured explicitly for the file type based on file extension pattern matching, IIS consider next whether the file and directory referenced in the request are executable for the impersonation security context active in the request, such as IUSR_MachineName or IWAM_MachineName if no alternative to the default is configured and any authentication used did not result in a change to the effective security principal and context, by examining its NTFS DACL. In addition IIS checks to see if execute permission has been granted in the Metabase for the item.

For an ISAPI extension referenced explicitly in the request, IIS loads the DLL into memory if not present already and uses the EXTENSION_CONTROL_BLOCK structure with the HttpExtensionProc DLL entry-point to invoke ISAPI processing. No new process is created, as the process model under which IIS host the active WAM application determines where the ISAPI loads. For executables that are not script engines with Application Mappings, the URL references the executable explicitly and may include the .exe file extension. These executables are treated as CGI programs and are launched by IIS in a new process for each request. The CGI can optionally accept information about the request from IIS by way of environment variables and standard input. Anything the executable writes to standard output is delivered by IIS to the HTTP client. The default behavior when none of these conditions is true is for IIS to treat the item as static content and serve it raw, as it exists in the static content file, with the appropriate Content-Type HTTP header so that the client can determine how best to handle the response.

IIS versions prior to 6 supported a registry entry CreateProcessAsUser that would override the default behavior where processing of CGI programs would occur using impersonation and the security context of the current thread. Instead, this registry value, when set to 0, would cause CGI programs to execute under the security context of the process rather than the thread, and thus ignore the active impersonation context, if any.

DWORD registry value: HKEY_LOCAL_MACHINE\SYSTEM\
CurrentControlSet\Services\W3SVC\Parameters\CreateProcessAsUser

Complete Process Isolation with WSH CGI

ISAPI filters should be deprecated in favor of Global Interceptors and ISAPI extension wrappers whenever possible. Filters are more difficult to secure and they get involved in request processing even when they are not needed. Rather than attempting to retrofit the ISAPI filter architecture to provide new health and hardening features, IIS 6 leaves the ISAPI filter model unchanged, and unimproved, compared to versions 4 and 5. In spite of the new enhancements to ISAPI extensions introduced with IIS 6 that add scalability, resistance to DoS conditions, and reliability while offering a migration path away from filters by chaining ISAPIs together, there are many instances where the right thing to do is deprecate ISAPI completely and move server-side application logic into the process-per-request Common Gateway Interface model that triggers an executable for each HTTP request.

Everything that happens on a particular thread in a particular process is vulnerable to malicious or errant actions taken by other threads of the same process. Under IIS 6, worker process recycling can occur in spite of the fact that many of the process threads are still healthy. While ISAPI boosts performance and capacity, its cost in terms of

application development and quality assurance manpower is very high, and the risk of introducing security bugs in compiled code should discourage the automatic adoption of the ISAPI architecture for many IIS deployments. The most secure mechanism available for extending the functionality of IIS through server-side application code is good old CGI because there is far less architectural complexity, fewer things that can potentially go wrong, and absolute isolation not just between applications but between each request along process boundaries. Very importantly, there is also minimal Microsoft code at work during request processing. Whenever an IIS box can meet its processing load and performance requirements during peak usage time without using ISAPI, consideration should be given to this alternative.

One of the most versatile and securable ways to implement server-side application logic that is isolated CGI-style in a separate process for each request is to configure the Windows Script Host (WSH) as an IIS script engine. With WSH configured to perform processing of requests for script files, the server incurs the overhead of loading WSH into memory and starting its primary thread of execution within the context of a new process for each request. This overhead is in some respects unreasonable for a busy server.

However, it takes far less manpower and programming talent to deploy a security-sensitive Web application based on CGI with WSH as the script engine than it does to build hardened ASP code or completely configure code access security policy settings for a service build upon the .NET Framework. For this reason alone many deployments should consider investing more money up-front in the acquisition of powerful server hardware and then hire a single system administrator with infosec knowledge and the programming ability of a power user, including the ability to write administrative scripts, to design, build, and also maintain all server-side application logic. With a single person or a small team responsible for all aspects of a Web application and its hosting platform you avoid the security problems that tend to show up in places where responsibilities overlap.

One of the security benefits of WSH is its new Software Restriction Policy first introduced with Windows XP. Configuring WSH as a script engine and leveraging its improved security features in the context of a Web application are simple tasks to accomplish.

Application Extension Mapping for The WSH Script Engine

The WSH script engine can be configured in IIS on a per-Web site basis or as part of the WWW service master properties setting the default Application Mappings for new sites. WSH comes in two varieties, the Microsoft Console Based Script Host (cscript.exe) and the Microsoft Windows Based Script Host (wscript.exe). The former is designed to never display popup windows, while the latter assumes that a human user wants to interact with popups for error conditions and other events. You should use cscript.exe not wscript.exe when configuring WSH as an IIS script engine. Figure 11-5 shows the Application Extension Mapping required for cscript.exe to work as an IIS script engine. The %s parameter following the cscript.exe full path instructs IIS to pass the .vbs script's PATH_TRANSLATED to WSH as a command line parameter when CGI processing is initiated to identify the script for WSH to load and interpret.

Selecting the Script engine check box enables the script handler to interpret script content from a directory without Execute permission. The Check that file exists check box tells IIS

to avoid invoking the script engine at all if the file requested by the client does not exist on the server. This prevents a request from arriving that names a bogus non-existent file in order to cause default processing to occur in the script engine code. Often times, as is the case with cscript.exe, running the program with no parameters results in usage instructions or other diagnostic output that may be useful to the attacker such as revealing the script engine name and version number.

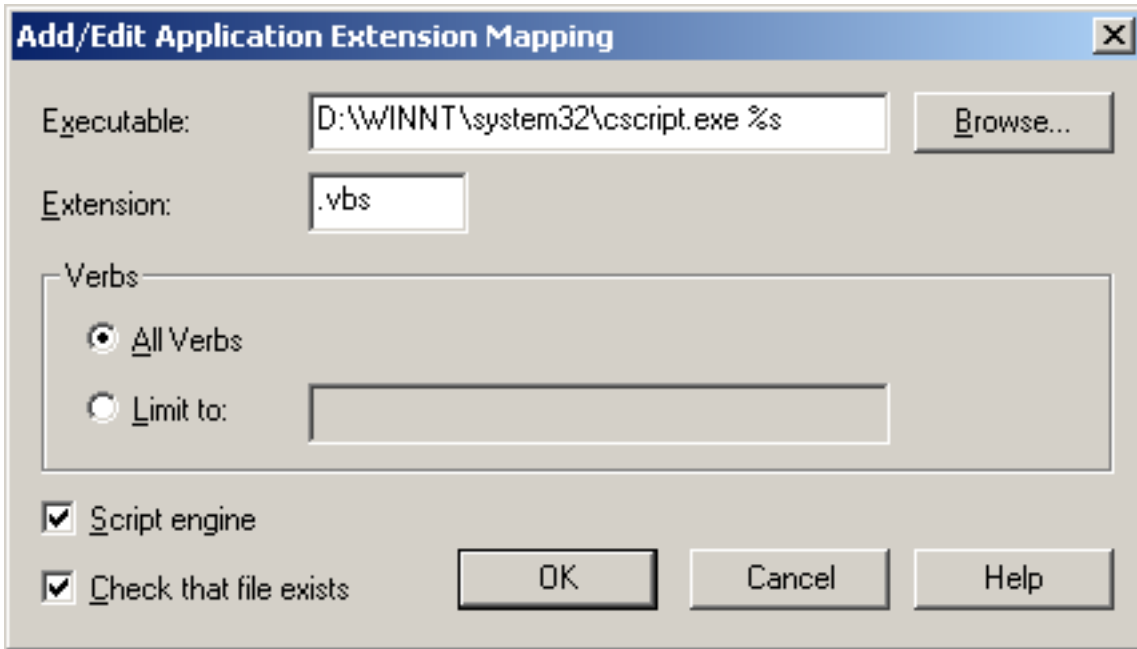


Figure 11-7: Configure Microsoft Console Based Script Host as a Script Engine for IIS

One extra step is necessary under Windows .NET Server with IIS version 6. Microsoft Knowledge Base Article Q311481 explains that under IIS 6 and .NET Server the default DACL settings for all executable programs in the System32 directory denies access to IUSR_MachineName for security reasons. To configure cscript.exe as an IIS 6 script engine you must first modify the DACL so that the impersonation account in effect during request processing of server-side WSH scripts has both Read and Execute access permission to the file.

Any script engine that conforms to the active scripting API and includes a generic script language interpreter in the form of a DLL can be used with IIS. Use the following instructions to script with Python instead of Visual Basic Script in ASP.

Knowledge Base Article Q276494 titled "Using Python Scripts with IIS" states:

Alternatively, you can use the Python interpreter as your script interpreter in your ASP pages.

After you have the Python scripting engine registered, create a file by using Notepad and include the following lines of code. Save the file in the scripts folder as Python.asp. See <http://www.python.org> for more information.

```
<%@LANGUAGE=Python%>
```

You should consider using a file extension other than .vbs that doesn't reveal anything about the script engine being used on the server. Ideally the file extension selected will have no

default application handler configured on the server or the extension will be mapped to something harmless like notepad.exe to prevent a potentially vulnerable executable from being launched locally through an instruction to open one of your server-side application files using the default open action configured in HKEY_CLASSES_ROOT for files with filenames that end in the specified file extension.

An innocuous file extension also keeps an attacker from making an educated guess about the workings of your server-side code by hiding clues to potential vulnerabilities, forcing the attacker to probe your server to gather intelligence that will identify a means of attack. Such probing can easily be captured by an integrated honeypot, enabling counter-intelligence and automated defenses.

HKEY_USERS\DEFAULT Hive for IUSR_MachineName with WSH

WSH relies on registry entries in the HKEY_CURRENT_USER hive to read its various configuration settings. Due to the fact that IUSR_MachineName and IWAM_MachineName or other impersonation accounts don't normally have user account Documents and Settings subfolders with NTUSER.DAT persistent hive files for the OS to load into HKEY_CURRENT_USER, the execution context of any CGI launched WSH instance uses HKEY_USERS\DEFAULT instead. For WSH to function properly as an IIS script engine you must add the following registry key:

HKEY_USERS\DEFAULT\Software\Microsoft\Windows Script Host\Settings

Give Everyone read access to this registry key and its parent. Every user account that does not have its own persistent NTUSER.DAT hive file stored in a Documents and Settings subfolder (or retrieved dynamically for domain accounts when Active Directory or NT Domain services are in use) will need permission to read these keys and the values they contain as a replacement for a customized HKEY_USERS hive. Now add the registry values from Table 11-2 under Settings.

Table 11-2: WSH HKEY_USERS\DEFAULT Registry Hive Values

Registry Value	Data Type	Setting
BatchMode	REG_DWORD	0
DisplayLogo	REG_DWORD	0
Timeout	REG_DWORD	0

The DisplayLogo setting is especially important because it suppresses the logo startup banner that WSH displays by default. Since the CGI execution model is designed to deliver everything a CGI program sends to standard output (STDOUT) over the network to the client as part of the HTTP response, the default WSH logo causes corruption of the HTTP headers and makes the browser unable to parse the response correctly. The WSH Version 5.6 logo startup banner looks like this:

Microsoft (R) Windows Script Host Version 5.6
Copyright (C) Microsoft Corporation 1996-2001. All rights reserved.

When WSH executes as a CGI program the first thing it should do is send HTTP headers followed by the double carriage return/line feed delimiter that separates HTTP response

headers from the HTTP response body. The following WSH script demonstrates the minimum requirements for such a CGI response.

```
WScript.Echo "Content-Type: text/html"  
WScript.Echo  
WScript.Echo "<HTML><BODY>Aloha World!</BODY></HTML>"
```

With WSH now configured as a script engine under IIS, your server-side scripts can do anything WSH can do. This is a bad thing, because WSH is actually capable of doing more than Active Server Pages or other script engines can do. To prevent abuse of WSH from within IIS either by coworkers or by malicious attackers you should make use of the security facilities for WSH including Software Restriction Policy settings and digital signatures for WSH scripts. All of the new security features provided by WSH 5.6 or later are now part of your IIS Web hosting platform and they can all be used to defend the box from harm by malicious server-side scripts.

ISAPI may be the Achille's heel of IIS. However, version 6.0 of ISAPI and version 6.0 of IIS give developers and administrators the ability to protect this weak spot while continuing to benefit from ISAPI performance. For everyone involved in building and deploying ISAPIs the complexity of securing and hardening filters exceeds that of extensions, and IIS 6 therefore doesn't bother to attempt improvements to the filter architecture. Instead, more manageable and more easily coded ISAPI extensions that were always designed to operate as application code rather than cooperate as global layered filter providers are Microsoft's future emphasis for ISAPI security.

Health and recycling parameters, performance and impersonation identity parameters, and dynamic reporting of unhealthy ISAPIs within IIS 6 worker process gardens and application pools are just some of the reasons that the new and improved ISAPI architecture justifies its continued use.

There are times, though, when you have to acknowledge that you aren't able to meet the increased challenge of secure ISAPI application development. Unlike server-side scripting and programming with the relatively security-hardened Microsoft .NET Framework when you build ISAPIs you must create the most complicated type of Windows-based compiled native code: a DLL that loads in the context of a network service; making the ISAPI the most challenging of code to properly secure.

For these times, whether they occur because of inadequate skill or whether they occur because of inadequate financial or human resources, it is better to avoid ISAPI and rely instead on the Common Gateway Interface (CGI) alternative. It is far better to build a secure system that wastes CPU time than to build an insecure but efficient system that causes harm to the people who mistakenly place their trust in it.

Chapter 12: Authentication Credentials

Credentials exist in order to enable authorization of an entity's permission to carry out a requested action or access certain resources. Authentication is never performed for its own sake. You could argue that maybe it should be; that every morning when we wake up we should all be authenticated to ensure that we still are who we think we are and that only we possess and control our unique identity. We all know that we didn't take any actions with our identity while we slept, therefore any actions that the world at large perceived us to have taken during that time period are absolute proof that somebody else has a copy of our identity and that we are no longer solely the person we think we are but now we are also the actions taken by another, at least in the absence of proof to the contrary. This sort of social risk, identity theft, is possible because authentication credentials we use to prove our identity are designed without provably-secure information security and without safeguards that enable easy revocation of compromised identifying information or credential reissue.

Any system that enables a person or a computer to prove to another person or computer the authenticity of the source of an instruction or request through the use of some type of credential is an authentication mechanism. Like the cryptographic algorithms upon which it is often based, authentication is either trustworthy and strong or it can be broken or fooled with trivial effort and tools. Bad authentication methods and algorithms can do more harm than good because they lull the uninformed into a false sense of security.

Credentials are central to the concept of authentication. The safety of credentials; resistance to forgery, brute force or dictionary attacks, and cryptanalysis when encoded or encrypted credentials are intercepted during transmission or while in a secure storage; is one aspect of an authentication method's real effectiveness. Another aspect is how credentials are protected from interception when they are used during an authentication event. In addition to these basic qualities of any authentication system there are human procedures and software tools for creating credentials and preparing a credential storage for later use during authentication events. The procedures and technical aspects of authentication produce either safe and effective protection against unauthorized activity or they result in the weakest link in a system's security.

Authentication is not just for HTTP in IIS. FTP, NNTP, and SMTP features of IIS also support authentication using essentially the same methods as described in this chapter with the exception of Forms Authentication and .NET Passport features. See Microsoft Knowledge Base Article Q324285 HOW TO: Set SMTP Security Options in the Windows .NET Server Family for more on SMTP authentication.

While there are numerous authentication methods possible in any Web application, and IIS in particular supports a variety of them, there are in practice only three meaningful options. The first is HTTP Challenge/Response authentication using the WWW-Authenticate header in an HTTP response to present an authentication challenge to the client in response to which the client sends an Authorization header in each subsequent HTTP request for the protection realm. This is how HTTP Basic Authentication works, for example. The second meaningful option is the use of HTML forms to prompt for and

receive authentication credentials, combined with an authentication token dropped as an HTTP cookie to the browser so as to associate each subsequent request from the same client with the authenticated session that resulted from successful forms authentication. Forms Authentication is used reliably and safely by ASP.NET and Microsoft .NET Passport so there is no need to reinvent the wheel. There are more insecure ways to create authentication using HTML forms than there are correct ways, so ASP.NET and .NET Passport are valuable models to study if you must implement your own Forms Authentication mechanism.

Microsoft .NET Passport is essentially a Kerberos version 5 adaptation for the Web that uses Forms Authentication and Forms Authentication tickets to achieve a minimal Kerberos-style authentication service via HTTP in a way that is compatible with platform-independent thin-client HTML user interfaces and Web applications. Microsoft .NET Passport is a ticket granting service similar to a Kerberos version 5 Key Distribution Center, and the Passport authentication store is the global Passport user database against which a user's plaintext credentials are matched. Credentials are never disclosed to a Passport-enabled Web site. Instead the Web site is given only the authentication ticket generated by the Passport service for the client to use as a trusted authentication token in place of credentials.

Finally, without a doubt the best authentication method available to Web applications, the SSL protocol offers mutual authentication so that both the client and the server possess public key and private key pairs with digitally signed certificates that establish trust in the authenticity of each party's identity and their associated public key. Asymmetric encryption is also used by SSL, with or without client certificates, to facilitate the exchange of the symmetric encryption key used for data privacy in the SSL protocol. The addition of a client certificate to the SSL session adds authentication of the requests sent by the client to the server in addition to the encryption and server identity verification features provided through the presence of the server certificate alone in the most common SSL usage scenario. Chapter 14 shows how to create and configure client and server certificates for use with SSL for data encryption and identity authentication.

While a number of other authentication options exist, most of them are unworkable or unsafe. In particular there are products built around IIS such as Microsoft Site Server that offer authentication methods that must never be used. Automatic cookie authentication, for example, was seriously flawed and could not be used as an authentication method. It was only marginally useful for personalization, as well, due to its difficulty preventing duplicate session identifiers from ending up in use by multiple users. If you must use an authentication method other than the three meaningful, reliable methods discussed here, be sure to conduct a thorough forensic security analysis of the system before you put it into production.

HTTP Challenge/Response WWW-Authenticate

The simplest type of authentication employed between HTTP clients and servers is an HTTP Challenge/Response exchange initiated by the server when it determines that authentication credentials are required before the client can be granted access to the resource or content requested. The server delivers HTTP error code number 401 "Unauthorized" and a challenge to authenticate that the client receives in the form of an

extra HTTP header, WWW-Authenticate. The client parses out this HTTP header and determines what, if anything, it can do to respond to the authentication challenge.

When an access attempt sent in an HTTP request to IIS results in a 401 “Unauthorized” error code, IIS sends the required WWW-Authenticate challenge header listing the types of authentication allowed for the requested resource. The HTTP client has the option of resubmitting the request to IIS including a response to the WWW-Authenticate challenge. When providing credentials in response to a challenge, HTTP clients send an Authorization HTTP header.

The Authorization response header can be sent by a client in an HTTP request without first receiving the WWW-Authenticate challenge from a server that requires authentication. This avoids the HTTP Challenge/Response round-trip with the server when the client knows in advance that a request requires an Authorization header. Sending an Authorization header in a request without first receiving a 401 HTTP “Unauthorized” response containing a WWW-Authenticate challenge is referred to as preauthentication. Common WWW-Authenticate headers are shown in Table 12-1.

Table 12-1: Standard WWW-Authenticate HTTP Header Values

Authentication Method	WWW-Authenticate
Basic	Basic
Windows NT Challenge/Response	NTLM
Digest	Digest
Integrated Windows	Negotiate, NTLM

In the case of Negotiate or NTLM, Internet Explorer may automatically send a response to the WWW-Authenticate challenge in a new HTTP request that contains Windows logon credentials without notifying the end-user or asking for permission to do so. By default the Local intranet and Trusted sites security zones are configured to allow automatic logon for User Authentication through HTTP Challenge/Response. Knowledge Base Article Q267850 entitled “Anonymous User Appears to Have Access Even When File Permissions Are Denied on Intranet” details automatic client Integrated Windows Authentication failover that occurs whenever possible, resulting in unexpected access to protected content without an apparent authentication step.

Preventing Automatic Logon Attempts via HTTP

Automatic logon attempts not only result in occasional confusion due to improper authentication settings and permissions on IIS files and folders but it is also a distinct vulnerability in the Windows platform as a result of the ease with which a remote attacker or anyone with physical access to a user’s LAN can compel Windows to attempt NTLM Authentication with network resources such as file shares or IIS. If an attacker in control of a remote Web site can trick IE into considering it to be part of the Local intranet security zone and IE has not been reconfigured to disallow automatic authentication attempts in the Local intranet zone then the remote Web site will automatically receive the NTLM hash of the user’s Windows password. Figure 12-1 shows how IE User Authentication can be switched from automatic to manual with a new end-user prompt for the first authentication event with each Web site.

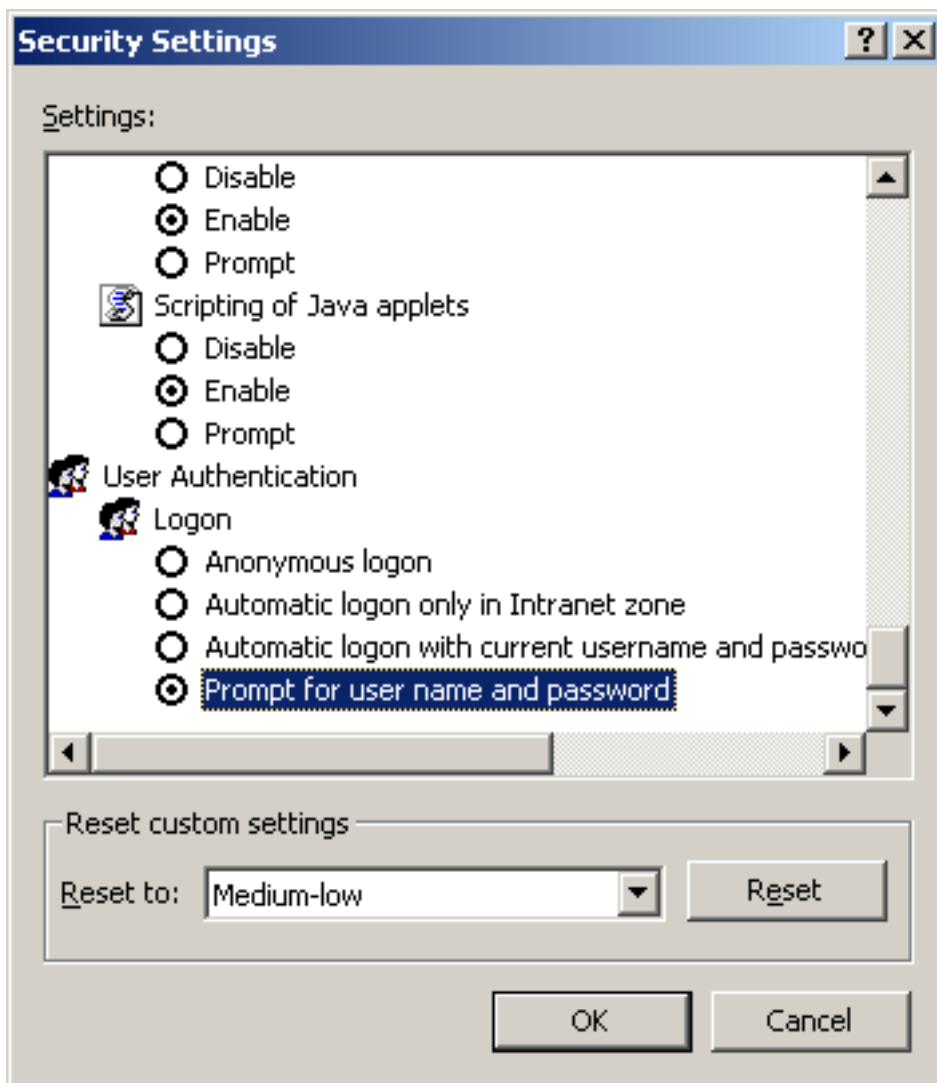


Figure 12-1: Override IE Default User Authentication to Prevent Automatic Logon

There are many other ways for authentication to occur in a Web application other than through use of the HTTP WWW-Authenticate header. IIS, Site Server, Commerce Server, and other IIS-based products and platforms including ASP.NET all provide variations on the authentication theme. Depending upon the security requirements and operating environment of a given application and Web site, and in particular who its users are and where they are located on the network, any of the authentication options provided by your IIS deployment may offer an appropriate level of security. With each authentication method there are tradeoffs and benefits, and while there is no universally right way to do authentication, it's important to avoid combinations of authentication method usage scenarios that are not compatible for security reasons. One such unacceptable combination is unencrypted HTTP Basic Authentication using WWW-Authenticate and Authorization Challenge/Response.

Unencrypted Basic Authentication with Base64 Encoding

Basic Authentication without SSL encryption, encrypted VPN, or IP Security causes plaintext authentication credentials to travel across the network where they can be intercepted

Q	R	S	T	U	V	W	X	Y	Z	a	b	c	d	e
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46
g	h	i	j	k	l	m	n	o	p	q	r	s	t	u
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62
w	x	y	z	0	1	2	3	4	5	6	7	8	9	+
		/												

When fewer than 24 bits are input as the final 3-octet input group, the equals sign (=) is used for padding to ensure that even the final input group transforms into an output of 32 bits (4 octets). At most a Base64 encoder will add 2 equals signs (==) to the end of encoded output in the case where the final input group contains only 1 octet of data to be encoded. For example, the following encoded Basic Authentication credentials represent “username” as the user name and “password” as the password.

dXNlcm5hbWU6cGFzc3dvcmQ=

Decoded, this Base64-encoded character sequence represents the 8-bit octets 01110101 01110011 01100101 01110010 01101110 01100001 01101101 01100101 00111010 01110000 01100001 01110011 01110011 01110111 01101111 01110010 01100100 or the ASCII values 117 115 101 114 110 97 109 101 58 112 97 115 115 119 111 114 100 which correspond to the ASCII string “username:password”. Any octet binary sequence can be represented using Base64 while incurring only a 30% increase in the size of the encoded data compared to its original encoding format.

The informational RFC 1945 was made obsolete by RFC 2616 and RFC 2617 when HTTP version 1.1 was defined as Internet Standard HTTP. The Authentication portions of the Internet Standard HTTP 1.1 protocol are now detailed in a separate RFC 2617 named “HTTP Authentication: Basic and Digest Access Authentication”. RFC 1945, RFC 2616 and RFC 2617 can each be found at the following URLs.

- <http://www.ietf.org/rfc/rfc1945.txt>
- <http://www.ietf.org/rfc/rfc2616.txt>
- <http://www.ietf.org/rfc/rfc2617.txt>

Basic Authentication relies on either the local users and groups database or on a Windows Domain as the default source of user accounts and groups. Any Windows Domain accessible to the server box can be used for Basic Authentication user account mapping. The user ID and password supplied by the HTTP client must match a user ID and password in the user database of the selected Windows Domain. When authentication succeeds, IIS construct an appropriate impersonation token through a call to LogonUser from the Win32 API. The Windows Domain selected, if other than the default or none if the server box is not part of a Windows Domain, is not disclosed to HTTP clients, but Basic Authentication includes the informal notion of a Realm that does display to the user as part of an authentication prompt.

When users authenticate to IIS with Basic Authentication, the availability of plaintext credentials enable the creation of a Windows platform security context and token through a call to LogonUser with LOGON32_LOGON_NETWORK specified. This call returns an impersonation token upon successful logon with the specified credentials. IIS provide the impersonation token to Win32 API security calls that require impersonation tokens such as COM+ cloaking or ImpersonateLoggedOnUser. This causes authentication to occur based on the information stored in the local user account database or the Windows Domain selected for the site. When Site Server Membership is used, authentication occurs against an LDAP directory known as a Membership Server. Commerce Server also includes this feature, implemented as an ISAPI filter. Under either of these IIS-based environments, the impersonation token is fixed at that of an additional impersonation account similar to IUSR_MachineName anonymous impersonation but used only in the context of requests that include valid Basic Authentication credentials. By default IIS rely on the local user database as the credential storage for authenticated user security token lookup during impersonation.

While a Windows Domain can be selected for each password protected Web site or resource by using the ISM MMC user interface, FTP sites can authenticate against a Windows Domain other than the default only by setting the String type Metabase setting manually: LMMSFTPSVC\DefaultLogonDomain as described in Knowledge Base Article Q184319

IIS cache the user security token that corresponds to the identity authenticated by way of the specified credentials so that lookup of the SID, privileges, and group membership of the corresponding Windows user account need not occur each time the same credentials are received from a client. In addition, caching of the impersonation token allows IIS to avoid redundant calls to the LogonUser Win32 API function that performs HKEY_USERS registry hive preparation from either the user's NTUSER.DAT file or .DEFAULT HKEY_USERS subkey and creates a new impersonation token through a Windows LOGON32_LOGON_NETWORK logon type. This makes it possible for a user to authenticate under both a new password and the old one for a period of time immediately following a password change. A registry value, shown below, controls the length of time for which IIS cache user security context impersonation tokens.

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\
InetInfo\Parameters\UserTokenTTL
```

The data type of the UserTokenTTL registry value is DWORD and its range is 0x0 to 0x7FFFFFFF indicating the number of seconds that tokens will be cached. Integrated Windows Authentication access tokens are not cached in this way so the TTL (time to live) value set in this registry value has no effect in Integrated Windows Authentication. The default setting for UserTokenTTL is 15 minutes, or 0x384, even though this registry value does not exist by default within the Parameters registry key.

Forcing Browser Clients to Reauthenticate

There is no provision within RFC 2617 for a Web server to instruct a Web browser to flush its stored password cache and prompt the user to reauthenticate. One of the most common reasons for Web developers to adopt an authentication mechanism other than Basic Authentication (with SSL, of course) or develop a custom authentication solution is that this omission is perceived as a severe security risk. Users can't be trusted to reliably

close all browser windows to force cached credentials out of memory so that subsequent attempts to access the password-protected resource will be met with a new authentication challenge. Further, there is no facility for expiring sessions on the server side with most Basic Authentication implementations, including that provided in IIS. Because full plaintext credentials are provided (Base64 encoded) to the server with each HTTP request, IIS perceives the request to be fully authenticated whether one hour or one week has passed by since the last time IIS received and processed the last request with the same Basic Authentication credentials. This undesirable side-effect of RFC 2617 for practical deployment has led to a work-around that is supported in newer browsers. The following embedded username and password syntax preceding an @ symbol and followed by the FQDN of the server to contact tells the browser what credentials to offer when challenged with a 401 "Unauthorized" HTTP error code:

```
http://username:password@FQDN
```

Redirecting the browser in this way to an HTTP URL with embedded credentials that are invalid for the requested resource forces reauthentication even in the case where the user has previously selected Save this password in your password list to store the specified credentials in a persistent password file. Because the new (failed) authentication attempt overrides the previous (successful) authentication that resulted in cached credentials on the client, the cached credentials are discarded. When prompting the user to reauthenticate, the browser will display the username embedded in the URL prepopulated in the entry field, requiring the user to reenter only their password. To avoid displaying the user's authentic username, an instruction can be displayed instead such as "ENTER YOUR USER NAME HERE" as shown below. The result of this URL syntax when used to access a FQDN path that requires HTTP Basic Authentication is a Basic Authentication prompt.

```
http://ENTER%20YOUR%20USER%20NAME%20HERE:password@FQDN
```

HTTP over SSL (https://) URLs may also be used with this syntax. You've probably used this syntax before in connection with FTP servers and ftp:// URLs. The fact that IIS can redirect the browser to such a URL as a means to force it to flush its cached Basic Authentication credentials and reauthenticate is a valuable addition to practical HTTP client/server implementations even if it never ends up becoming an official part of the HTTP specification. Actually, Uniform Resource Locators syntax is not even part of the HTTP specification in the first place, having been defined by an IETF working group document of its own: RFC 1738. Embedding credentials in http:// or https:// URLs wasn't done until recently as a result of a need to solve this credential cache problem, but the //<user>:<password>@<host>:<port>/<url-path> syntax was always part of the URL specification defined by RFC 1738.

Another interesting, if somewhat impractical, suggestion is made in Microsoft Knowledge Base Article Q195192 HOWTO: Clear Logon Credentials to Force Reauthentication for building an ActiveX control to clear Basic Authentication credentials from cache and force the client to reauthenticate with the server.

For an interesting discussion of another possible application of embedded user credentials in URLs, see Microsoft Knowledge Base Article Q281408 entitled "How to Implement a

Single Logon Across Multiple Web Servers”. This article proposes that a Web site supporting Basic Authentication can facilitate seamless transitions between it and other trusted Web sites that rely on the same authentication store, or where a Web application stores Basic Authentication credentials for remote sites on behalf of the authenticated user, by redirecting the client to a URL that includes the credentials required by the remote HTTP server. This concept is termed “forwarded credentials” and in the context of SSL-secured cooperating Web sites it has merit. There are also a couple useful recommendations in this Knowledge Base article for protecting against credential leakages by way of shoulder surfing or browser history cache when credentials are embedded in the URL if you do employ this technique.

Basic Authentication Support in .NET Framework Classes

One of the primary advantages of Basic Authentication over all other types is that it has widespread support in client software and development tools. While it is technically possible for any authentication mechanism to be supported in any software, the reality is that only Internet Standard HTTP 1.1 Basic Authentication has universal support across platforms, Web browser versions, proxy servers, and end-user or company-wide security policies today. Even HTML forms authentication which is second-best in terms of widespread support isn’t offered as an authentication type in most development tools or end-user software programs other than Web browsers. The integration into HTTP that Basic Authentication provides combined with its early introduction have made it the authentication method of choice in spite of its inherent insecurity at both ends and all along the communications path between client and server. While HTML forms authentication is better for end-users, and it is preferred by Web designers, because of the convenient Web page user interface branding integration it enables, WebDAV access to publishing points (see Chapter 15) and other developer-oriented or administrative tasks require Basic Authentication support rather than HTML forms. Microsoft .NET Framework Class Library classes provide abundant support for Basic Authentication from within network client code.

Every .NET class that supports Basic Authentication implements the protocol defined in RFC 2617. There are two classes in particular that serve as the foundation for Basic Authentication credentials in Microsoft .NET and they both implement the same interface: `System.Net.ICredentials`. Individual credentials are stored in instances of the `System.Net.NetworkCredential` class, and collections of credentials are stored in `System.Net.CredentialCache` objects. These classes can be used interchangeably because they both implement the `ICredentials` interface that is required by classes that actually make use of credentials during authentication events. The next two sections show how these `ICredentials` classes are used in typical .NET client code to construct and send an authenticated HTTP request to a password-protected HTTP server and manage collections of credentials for different realms with a `CredentialCache` object.

Sending an Authorization Header in HTTP Requests with .NET

Basic Authentication is supported in the .NET Framework Class Library for both SSL encrypted and unencrypted HTTP requests. Whether you need to contact a password-protected Web site (your own, perhaps, for administrative or security audit purposes) from within ASP.NET, a CGI, or other program built on .NET Framework, managed code

required to add Basic Authentication credentials to an HTTP request is simple. The following source shows the steps required when System.Net.WebClient is used to send the authenticated HTTP request. Note the URL shown uses https:// and WebClient to establish an SSL connection before username and password are sent to the server. The credentials are therefore sent encrypted rather than as plaintext that can be intercepted easily and decoded using a simple Base64 decoder.

```
byte[] buf = null;
string url = "https://localhost/private/data.aspx";
NetworkCredential c = new NetworkCredential();
c.UserName = "username";
c.Password = "password";
WebClient http = new WebClient();
http.Credentials = c;
try { buf = http.DownloadData(url); }
catch(Exception e) {}
```

The ICredentials interface implemented by the NetworkCredential class defines a method, GetCredential, that is called by the WebClient object when it encounters a 401 “Unauthorized” HTTP response and WWW-Authenticate challenge. The result of the code shown here is not preauthentication but rather an initial HTTP request without an Authorization header, to which a password protected resource served by HTTP will presumably respond with a Basic Authentication challenge. The .NET Framework sends, in response to the challenge for authentication credentials issued by the server, the required Authorization HTTP header shown below.

Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=

Basic Authentication should never be used without SSL unless security is of no concern for a Web site or its users. One example of a scenario in which it is acceptable to use Basic Authentication without SSL is where nothing other than personalization of dynamic Web content or data mining preferences are stored on the server using the specified credentials.

Special caution must be taken, however, to inform users of such a password-protected personalized Web site that encryption is not used to protect their credentials because nothing confidential will ever be stored on or transmitted to the Web site. Many users violate strong password practices by using the same password in many places, and using Basic Authentication without SSL in a situation where the Web site has nothing of value to protect will expose such users to theft of the credentials they rely on for security elsewhere.

Protection Realms and Credential Cache

Part of the Basic Authentication specification calls for the server to specify the Realm name of the protected resource being requested when sending HTTP challenge WWW-Authenticate headers. The Realm is for informational purposes only, and browsers are not designed to associate different FQDNs and URI paths as belonging to the same protection realm based on matching Realm values. However, the Realm is an important

part of Basic Authentication because it is the only user interface customization possible with this authentication method. Web browsers display the Realm supplied along with a WWW-Authenticate challenge when prompting the end-user to enter credentials. End-users should not trust the Realm as a means of identifying the server to which they are attempting to login, but there is no mechanism to prevent them from doing so or being fooled into doing so.

Browsers prompt for authentication credentials whenever they encounter an HTTP 401 "Unauthorized" response with a WWW-Authenticate Basic Authentication header, even when the item being requested is an image embedded in an HTML document. This is one of the most problematic aspects of practical security, that is the actual level of protection against credential interception achieved in practice, with Basic Authentication. A browser can be made to prompt for authentication credentials a second time for a protection realm under control of the attacker without raising any warnings or causing end-users concern simply by throwing a cross-site scripting attack against an SSL-secured Web server. A typical Basic Authentication prompt that clearly lacks the information necessary for the end-user to decide whether or not the prompt is trustworthy. Even SSL fails to prevent such an attack through server identity deception and concealment. The unsuspecting user who enters their credentials a second time reveals them to the attacker without experiencing any functional change in the Web site they're using, and will never know that they responded to an authentication prompt once for the authentic server and once for a server belonging to a malicious third party.

The untrustworthy nature of the Realm value supplied by the server is an architectural flaw in Basic Authentication because end-users will never understand that this value has no meaning. Browsers should be redesigned so as to display the FQDN, IP address, and details from the SSL certificate if applicable instead of the Realm when prompting for Basic Authentication credentials. However, that's not the way things are in the real world today, so any exploit that allows an attacker to inject a simple tag that references a FQDN that the attacker controls is sufficient to result in leaked credentials. There are numerous ways for such attacks to circumvent warning messages even when SSL is employed, and the fact that the authentic server never sees the extra requests sent by clients to the malicious FQDN means that a minor Web site defacement or XSS exploit is only detectable from outside the network through security assurance crawlers designed to validate the output of Web applications and discover attempts to inject HTML or script into the first contact a client has with the server. Features such as HTML frames, style sheets, and new XML features make for fertile ground in which black hats can always find novel techniques to influence Web browser and Web server behavior in subtle ways. Still, these problems are only slightly improved by other authentication mechanisms.

One option for improving the security in practice of Basic Authentication to prevent credentials from ever leaking to third party servers as a result of XSS or tampering with Web page content sent to clients by Web applications is to enlist the help of the Microsoft .NET Framework for conducting authentication automatically with controls that avoid the problem of end-user deception. It's easy to put into code the knowledge and awareness that an HTTP Realm is meaningless, and that credentials should only be sent to servers that are authorized to receive them. It's more difficult to train all end-users so that they are resistant to the tricks and deceptions possible within the context of a Web browser client application. To this end, classes in .NET that support Basic Authentication such as

WebRequest and WebClient from the System.Net namespace provide AuthenticationManager support. The AuthenticationManager class is static and its purpose is to keep a list of all registered authentication modules, allowing .NET applications to register new ones. Each registered authentication module implements the IAuthenticationModule interface, and whenever authentication is required in response to WWW-Authenticate the AuthenticationManager static class is used by the .NET Framework to iterate through each registered AuthenticationModule, calling the Authenticate method on each of them until one of the modules returns an Authorization object with the necessary credentials to use in an HTTP Authorization header. In this way any .NET managed code can implement any required client-side credential cache mechanism.

Objects inherit from Authorization in order to be part of authentication modules used by AuthenticationManager. A public property of the derived class, Message, contains the Authorization string sent in a request to authenticate with the specified Realm. Another public property of the derived class, ProtectionRealm, is normally used for matching only URI parts including FQDN and path but a custom AuthenticationManager class could be used to extend this functionality to examine the Realm supplied as part of a server's WWW-Authenticate challenge as well as other data elements. The Basic Authentication Authorization object provided in the .NET Framework handles Base64 encoding of user name and password on behalf of a client application that supplies credentials. A CredentialCache object can be used to store collections of Authorization objects as shown in the following code.

```
CredentialCache cc = new CredentialCache();
byte[] buf = null;
string url = "https://localhost/private/data.aspx";
string url2 = "https://localhost/private/data2.aspx";
NetworkCredential c = new NetworkCredential();
c.UserName = "username";
c.Password = "password";
cc.Add(new System.Uri(url), "Basic", c);
NetworkCredential c2 = new NetworkCredential();
c2.UserName = "username2";
c2.Password = "password2";
cc.Add(new System.Uri(url2), "Basic", c2);
WebClient http = new WebClient();
http.Credentials = cc;
try { buf = http.DownloadData(url); }
catch(Exception e) {}
```

The previous code sample demonstrated passing explicit credentials to the WebClient object whereas this sample shows the creation of a CredentialCache that is passed to WebClient instead. The WWW-Authenticate header indicates the type of HTTP authentication supported by the URI and the first matching Basic Authentication credential set is selected from the CredentialCache. The process illustrated here is similar to that which a typical Web browser client implements with the exception that there is no automatic prompt for credentials displayed to the user. It is this automatic prompt and potentially misleading Realm display name, information that in reality is

inadequate to enable the user to determine whether or not it is safe to supply credentials in the first place, that can be avoided with custom client code. Basic Authentication is only flawed procedurally in this one respect, although it is also flawed in its requirement for the transmission of actual credentials. Even when SSL encryption is used, it is less desirable to transmit full credentials than it is to transmit a hash of those credentials because the server has to have a copy of the full credentials in order to validate them at runtime. This exposes the credentials to theft if the server is ever compromised. It is far better for the server to store only a hash code generated based upon a user's credentials, and thereby avoid transmitting or storing full credentials at any time. Digest Authentication implements just such an improved authentication algorithm making it superior to Basic though not as widely supported.

Basic Authentication Support in Windows HTTP Services (WinHTTP)

Windows HTTP Services (WinHTTP) are a client-side API for sending HTTP requests programmatically. The response sent by the HTTP server is received and optionally processed as text by the code that calls into WinHTTP. WinHTTP version 5 and later are available only for Windows 2000/XP/.NET and version 5.1 has important security enhancements like the ability to abort communication with an SSL-secured server when its server certificate fails verification. Version 5.1 is also the first version of WinHTTP that won't attempt to authenticate by default when presented with an authentication challenge. WinHTTP version 5 introduced support for Automation, making it possible to call into WinHTTP from script. The following sample code shows a simple Visual Basic Script that creates an instance of WinHTTP 5.1 using CreateObject. The sample code works with Windows Script Host, but the Automation-compliant scriptable interface exposed by WinHTTP works with any active script-compatible host environment including Active Server Pages. Notice the call to SetCredentials where Basic Authentication credentials are specified.

```
dim req
Set req = CreateObject("WinHttp.WinHttpRequest.5.1")
req.Open "GET", "http://FQDN", false
req.SetCredentials "username", "password", 0
req.Send
WScript.Echo req.ResponseText
```

Unlike the Microsoft WinInet API, which was the previous client-side HTTP request API for Windows platforms, WinHTTP is designed to be thread-safe and robust when used in a services environment. It is safe to use WinHTTP as part of a Web application or heavy-lifting administrative tool. WinHTTP also exposes a COM interface so that it can be used from any Windows software. With SSL support built into WinHTTP, and its support for explicit authentication and other communications features required by server-to-server communications, this component is a valuable addition to the Windows Web Services environment. WinHTTP also supports client certificate authentication as shown in Chapter 14 and .NET Passport authentication, making it the first programmatic client support for Passport and thus a potentially handy tool for the kiddies to use to brute-force Passport credentials.

Server-Side User Token Cache Settings

By default IIS are configured to cache the platform security token for authenticated user accounts or the corresponding impersonation accounts. Whether or not the platform security token is cached has no impact on the client-side decision to send Basic Authentication credentials in a request that the client deems to require them. And IIS won't issue a new 401 "Unauthorized" HTTP result code with authentication challenge WWW-Authenticate header when the platform security token used for access control on the box isn't available in cache, IIS simply obtain the necessary token over again through making the necessary Win32 API calls. It is important to be aware of this user token cache, however, because changes to impersonation settings and other platform reconfiguration events can result in unexecpted behavior if the cache is not properly flushed before subsequent authentication events occur.

A token cache timeout interval countdown starts for each new authentication event based on the value of UserTokenTTL at the time of authentication. Changes made to this registry value take effect only for new authentication events. See the following Microsoft Knowledge Base Article for more details on the user token cache interval: Q152526 Changing the Default Interval for User Tokens in IIS.

To manually purge the user token cache you can restart the IIS Admin service. A related cache worthy of note controls whether or not security descriptors (DACLS) are cached for files accessed by IIS. The following registry value also located under the Parameters registry key can be set to DWORD value 0x1 to turn on DACL cache.

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\
InetInfo\Parameters\CacheSecurityDescriptor

With DACL caching active, IIS never read files' DACLS a second time from disk after the files are stored in IIS cache. This is useful for security purposes because it prevents unauthorized permissions changes from taking effect until IIS restart. The DACLS encountered by IIS at startup, when each file is first accessed and its contents and DACL cached, are the DACLS that are used by IIS to enforce cached file access control. By default CacheSecurityDescriptor is disabled and IIS reread DACLS from disk for each file stored in cache each time the cache is accessed processing a request.

Microsoft Windows Digest Authentication

Digest Authentication functions just like Basic Authentication with respect to the HTTP Challenge/Response mechanism. Instead of specifying Basic as the type of authentication, Digest is specified in WWW-Authenticate and Authorization headers. The benefit of Digest Authentication over Basic is that plaintext credentials are never transmitted across the network from client to server. A client that supports Digest will compute an MD5 hash code of the user's credentials and transmit the hash to the server rather than the credentials themselves. The server, in turn, need not place a copy of the user's credentials into its authentication store, which provides some protection against credential theft when the authentication store is compromised by unauthorized access. The amount of time required to brute-force MD5 hashed credentials is substantial unless the passwords are weak and can be matched against a dictionary. In most cases this gives the server administrator an opportunity to replace compromised credentials before

an attacker can gain unauthorized access to any user account for which credentials will eventually be discovered through brute-force cryptanalysis.

While it is always useful to the security of a password-protected system for that system to store hashed credentials rather than plaintext credentials just in case the credential storage is ever compromised, the protection this offers against a sufficiently prepared and well-equipped attacker is minimal. Given enough advance computing preparation, an attacker can produce a chosen-plaintext dictionary of hashes for every possible password value.

Rather than attempting to brute-force a password that has been MD5 hashed by iterating through every possible password value and applying the MD5 hash algorithm to these chosen plaintext values one by one at runtime until a match is found, an attacker in possession of a comprehensive MD5 hash dictionary can do a simple database lookup to discover the plaintext user password. No Such Agency probably has such hash dictionaries, and if they're smart (there is empirical evidence to suggest that they are smart) the NSA also has keyed hash password dictionaries and ciphertext dictionaries or are busy producing them through the application of enormous computing power. Because passwords are constructed from subsets of known character encodings and are usually of limited length, any password storage can be cracked. An Encrypted or keyed hashed authentication store to which only the server that manages the store has access is far better than a simple hashed credential storage but even this extra protection is not uncrackable.

However, it is the best option available to remediate the security of password-based authentication systems. It would be far better to eliminate passwords entirely in favor of keys and properly-designed certificate-based authentication, but it will be some time before every legacy password is retired, and keys have problems of their own especially when trust is established through certificate chains. Keys are also unwieldy compared to passwords, and the practical risk/reward limitations hold back the complete phase out of password-based authentication.

Windows Digest Authentication only works in Windows Domains and the only Web browser that supports this authentication method is Internet Explorer 5.0 or later. The security enhancement afforded by Digest versus Basic Authentication is substantial even when SSL or IPSec are not used to encrypt the TCP packets sent and received during each HTTP session. When IIS issues a Digest challenge it supplies a nonce, a one-time-use value, that is randomly selected for the authentication event. To send a valid response and thereby authenticate with IIS, Internet Explorer computes a hash of the user name and password supplied by the end-user along with the nonce, the HTTP method, and the current URL. The time and processor cycles required to precompute hash dictionaries for every possible nonce, credential, method and URL combination used by your Web site is prohibitive even for the NSA. This makes the Digest hash sent over the network by the client even safer from cryptanalysis than the cryptographic protection applied to the Windows Domain Active Directory authentication store itself.

For The Nonce Unto Then Anes

The term nonce originated in the 13th century as a grammatical error. Speakers of Middle English weren't often literate enough to spell properly so statements from popular Middle English vernacular such as "to then anes" which meant "for the one purpose" evolved into single words, as "nanes", and thus unto the "nonce". The word is admired by Cryptographers, and its technical meaning can be flexible but revolves around the idea that a nonce is used only once, for one purpose, or as a characteristic of a single occasion or event such as a distinct Challenge/Response authentication.

Digest Authentication is implemented within IIS as part of a subauthentication DLL (IISUBA.DLL). This DLL must be present on the domain controller in addition to the IIS box, as it is required as an Active Directory extension to assist in carrying out Digest Authentication events brokered by IIS. Each user account stored in Active Directory must also be configured to store password using reversible encryption. This option is selected during Windows account creation or modification when Active Directory is used in a Windows 2000/.NET network deployment. For a given URL and request method, a given set of credentials will always result in the same MD5 hash code when IIS supplies the same nonce value. Further, an attacker who captures both the nonce supplied by IIS and the MD5 hash code supplied by the client in its authentication response to the Digest challenge could, with enough computing power and time, succeed in discovering the user's credentials through cryptanalysis with a simple brute-force chosen-plaintext attack.

This makes Digest Authentication vulnerable to replay attacks if and when IIS reuses nonce values. For this reason it is still important to use SSL or IPSec for privacy whenever possible.

Integrated Windows Authentication

Like Digest Authentication, Integrated Windows Authentication passes only a hash code from client to server rather than full credentials. This creates an interesting problem for IIS when it attempts to authenticate with network resources using the impersonation security context of the authenticated Integrated Windows user account identity. Because IIS do not have knowledge of the actual credentials, authentication with other password-protected systems by way of Integrated Windows Authentication or another method is impossible unless a trusted network authentication service can vouch for the identity and trustworthiness of the impersonation context. This authorized relay of credentials on behalf of the end-user, essentially double-hop authentication, is known as delegation. For example, if server-side Win32 API WNetAddConnection2 is used to add a network connection on the server in the context of an impersonated identity, IIS must be able to supply username and password in the API call or else new connection mapping fails. The first hop is from the end-user's Web browser to IIS and the second hop, where delegation occurs, is from IIS to the network service accessible to the IIS box. The issue of delegation has driven the evolution of Integrated Windows Authentication from its original LAN Manager (LANMAN) mechanism to the Windows NT enhancement (NTLM) and finally to support for the Kerberos version 5 (RFC 1510) authentication service.

When IIS is configured to require Integrated Windows Authentication two WWW-Authenticate headers are sent, the first specifying Negotiate and the second specifying NTLM. Only Internet Explorer 5.0 and later support Kerberos currently, and when IE receives a Negotiate authentication challenge along with an NTLM challenge it always responds to

both, letting IIS decide which method to use. This means that NTLM always goes out over the wire even when Kerberos is the method that both IE and IIS agree to use for authentication.

Integrated Windows Authentication is preferred by Internet Explorer over Basic Authentication. IE will attempt Integrated Windows authentication whenever it is an option provided by IIS and will use the currently-cached Windows logon credentials before prompting the user for a different user name and password.

Prior to the introduction of Kerberos support for Windows 2000 and IE 5, the inability for NTLM authentication to do delegation led to a number of work-arounds that could impose an additional authorized security context within concurrent impersonation sessions, the initial impersonation performed by the IIS application and the delegated impersonation session performed by another service. A common technique was to hard-code an additional set of authentication credentials for a sufficiently-privileged user account into the code that must access another password-protected resource while acting on behalf of the authenticated Web site user and running in that user's impersonation context. Another common solution was to place code that must execute outside of the scope of the active impersonation context and even outside the scope of COM+ cloaking inside a COM+ application (or MTS package under IIS 4) configured to execute under a fixed user account rather than accept impersonation instructions by way of IIS 4/MTS integration support or COM+ cloaking. For other deployments it was acceptable to configure a more privileged anonymous impersonation account and direct Web site users to a script, ISAPI extension or CGI program accessible without authentication. With a privileged anonymous impersonation account and no authentication required, IIS receive the necessary access rights for the password protected network resource while the rest of the Web application operates as normal. These techniques can still be used today if NTLM authentication is required for backwards compatibility with clients that don't support Kerberos version 5.

Windows NT Challenge/Response appeared under the Directory Security tab of each Web site folder or file properties page in Internet Service Manager for IIS 4. This setting is a simple combination of the Windows NT LAN Manager protocol (NTLM) with HTTP Challenge/Response authentication using WWW-Authenticate and Authorization headers as described in the previous section. Beginning with IIS 5 and Windows 2000, however, the terminology changed to Integrated Windows Authentication. The reason is that there are now a variety of different modes of operation possible for built-in authentication under Windows. If you still use IIS 4 or if you must provide legacy support to clients that used the old method of Challenge/Response Windows authentication you should be aware that security flaws in the original Challenge/Response protocol made it unsafe and you should switch to Version 2 of the Windows NT LAN Manager (NTLM) protocol. An additional concern worthy of note is that originally NTLM was not compatible with proxy servers, so simply requiring NTLM authentication was enough to prevent an external Internet-originated request from authenticating with an intranet Web server or accessing intranet-only content served by an IIS box that also served content to the Internet by way of a reverse proxy server. Most newer proxy servers now support NTLM authentication, however, and you can no longer assume that Integrated Windows Authentication implies access from the intranet only.

Windows NT LAN Manager (NTLM) Version 2 Authentication

Prior to Windows NT Service Pack 4 and Windows 2000, integrated Windows authentication supported two different and equally pathetic network authentication protocols: LAN Manager (LM) and its backwards-compatible update for Windows NT (NTLM). Both of these protocols failed to provide protection against cryptanalysis on intercepted credential hashes. The average personal computer was always more than enough processing speed for an attacker to discover plaintext authentication credentials when given an encoded hash from an intercepted LM or NTLM Challenge/Response authentication event.

Knowledge Base Article Q239869 How to Enable NTLM 2 Authentication for Windows 95/98/2000 and NT provides instructions on legacy system remediation. Legacy operating systems can be hardened against NTLM use as explained in Knowledge Base Article Q147706 How to Disable LM Authentication on Windows NT. Windows 2000 has a new disable NTLM security policy setting.

An interesting problem emerged under Windows 2000 after the ability to disable NTLM was implemented, COM+ applications configured to execute under specific Windows user account security contexts stopped working because COM+ 1.0 required NTLM and was unable to authenticate using the specified user account credentials with NTLM disabled.

Details can be found under Microsoft Knowledge Base Article Q275482 entitled "FIX: COM+ 1.0 Catalog Requires NTLM-based Authentication". Windows 2000 Service Pack 2 resolved this problem with COM+ 1.0 but the scenario it represents deserves more scrutiny. To appreciate the implications of this bug and its required fix you have to consider that Microsoft's quality assurance process missed this rather obvious incompatibility. This suggests that several years went by from the point of a Microsoft developer first coding the ability to disable NTLM in Windows 2000 to the point of first discovery that when this feature was actually used in the real world it caused COM+ to break. You've no doubt encountered similar phenomena where a vendor's products are shipped with a new feature that ends up having a bug proving beyond any doubt that nobody ever actually used the feature before it was shipped to customers. Insufficient quality assurance and minimal infosec forensic analysis is typical of software development practice across the industry, and it is one of the reasons that you must never presume that any software is safe to use, even for its intended purpose, in your deployment. Every feature and configuration option that you use should be carefully logged in a forensic notebook so that you can compare detailed technical analysis of features and quality assurance reports provided to you by vendors with your minimum security presumptions in the parts of your computer systems that have security dependencies.

Active Directory and Kerberos Authentication

IIS 5 with IE 5 or later now support Kerberos authentication using Negotiate headers under Windows 2000 and later. IE 5 versions default to Kerberos for Integrated Windows Authentication while IE 6 defaults to NTLM. With all versions of IE, OS versions prior to Windows 2000 do not support Kerberos. Therefore IE defaults to NTLM even if Kerberos Negotiate WWW-Authenticate is received from IIS by IE running on a Windows OS

version that is not Kerberos capable. Because delegation is possible only when Kerberos is the selected authentication mechanism for an HTTP session initiated by IE, you must take care to force IE to default to Kerberos. Since both Negotiate and NTLM authentication challenges are sent by IIS and responded to by IE, ensuring that Kerberos is used can be a challenge of its own.

Knowledge Base Article Q299838 Unable to Negotiate Kerberos Authentication After Upgrading to Internet Explorer 6 provides more information about default Integrated Windows Authentication settings and Kerberos under IE 6 and later. Knowledge Base Article Q277741 Internet Explorer Logon Fails Due to an Insufficient Buffer for Kerberos details the limits on number of Windows groups to which a user account may belong to avoid DoS conditions in Winlnet APIs. See Knowledge Base Article Q269643 Internet Explorer Kerberos Authentication Does Not Work Because of an Insufficient Buffer Connecting to IIS.

For Kerberos to be used for authentication between IIS and IE, both client and server must have access to an Active Directory Services-compatible Key Distribution Center (KDC). It is the KDC that receives requests for authentication and dispenses authentication tickets that can be used to verify the identity of the client and server. RFC 1510 defines the Kerberos version 5 specification. To make sure that IIS will utilize Kerberos when available as part of Integrated Windows Authentication, the following metabase property should be set. This property can contain multiple authentication methods separated by commas.

W3SVC/NTAuthenticationProviders = Negotiate

When multiple authentication methods are allowed for a password-protected resource, IIS send multiple WWW-Authenticate headers and it's up to the client to determine which of the allowable methods it prefers. Different clients decide authentication method preference in different ways, and different versions of IIS have changed the way that WWW-Authenticate headers are delivered such that certain Web clients behave differently when communicating with different versions of IIS. Internet Explorer is designed to prefer the authentication method listed in the first WWW-Authenticate header supplied by IIS, and originally, in IIS version 1.0, when both Basic and NTLM authentication was allowed for a protected resource IIS would send a WWW-Authenticate: Basic header first. IIS 2.0 changed the order so that NTLM was listed first. IIS 5 and 6 now supply the Integrated Windows Authentication header first, making it the preferred authentication method for IE when an IIS response indicates that Windows Authentication is available.

Anonymous Authentication

When Allow Anonymous is enabled, a number of subtle authentication dynamics emerge that aren't immediately apparent. First of all, when anonymous access is allowed to an application or file hosted by IIS, the anonymous impersonation account is the initial security context used in request processing. IIS will only deliver an authentication challenge in the event that the anonymous impersonation account lacks sufficient authorization privileges to carry out the requested operation or read the specified file. For this reason it is important to restrict access to password-protected content both through

the ISM MMC (or Metabase) and through NTFS DACLs. In the event that one is inadvertently or maliciously changed to allow anonymous, the other will still prevent access to the anonymous impersonation account security context. Every Web site, virtual directory, and file can have its own anonymous impersonation account setting. ISM MMC can be used to configure an impersonation account other than the default IUSR_MachineName for anonymous access.

Whereas Basic Authentication supports delegation, double-hop authentication, and NTLM does not because credentials are never transmitted from client to server as plaintext or even as ciphertext that can be decrypted, Anonymous Authentication has similar problems with anonymous delegation. UNC shares can be accessed from the server using the anonymous impersonation account for a Web site only if the share allows Guest access, the built-in Everyone group (SID equal to S-1-1-0), or the IIS box is a member of the same Windows Domain as the box that exposes the UNC share and the anonymous impersonation account is granted access permission explicitly, or implicitly through a group membership, in the resource's DACL and the remote server's local security policy settings. Distributed COM (DCOM) services operate similarly with the exception that DCOM doesn't automatically map unknown SIDs to the built-in Everyone group. This is true even if DCOM permissions grant privileges to the built-in Everyone group. A DCOM service must be able to identify the SID of the user as a known user for security reasons. This is a subtle difference of opinion between Windows file sharing and DCOM as to the meaning of "Everyone", where UNC shares allow completely anonymous guests and DCOM allows everyone who can be identified by a valid security descriptor (SD).

Another subtle delegation phenomenon arises when IIS lose control of the login password for the anonymous impersonation account. When you switch to an explicit administrator-controlled password instead, or use an impersonation account other than the default IUSR_MachineName and enter the password for the alternative impersonation account manually through the ISM MMC, it causes IIS to gain the ability to delegate authentication because IIS caches the impersonation account credentials explicitly when they are established manually for the impersonation account. When IIS control the credentials for the impersonation account, the same subauthentication DLL (IISUBA.DLL) used to process Digest Authentication logon events validates the password supplied by IIS and informs Windows that the password is valid. This prevents IIS from accessing certain password-protected UNC shares and other resources because of inability to delegate authority for the impersonation account, which requires IIS to respond automatically to double-hop authentication prompts. Because a call to IISUBA.DLL is the only way for IIS to authenticate to Windows on behalf of an impersonation account for which it controls the password, IIS are unable to perform delegation unless the administrator removes IIS control of the impersonation account password and explicitly configures a password manually through the ISM MMC instead.

When IIS control the impersonation credentials, a call to IISUBA.DLL results in a network logon through a call to LogonUserEx with the flag value LOGON32_LOGON_NETWORK. Under Windows NT and IIS 4 the alternative call path when IIS aren't allowed to control the impersonation account credentials is a call to LogonUserEx with the flag value LOGON32_LOGON_INTERACTIVE which results in a local logon rather than a network logon. The impersonation account under IIS 4 thus has to be granted the "Log on locally"

right when IIS don't control the account's credentials, and "Access this computer from the network" rights are sufficient otherwise.

Under Windows 2000/XP/.NET a new logon type flag value is introduced, LOGON32_LOGON_NETWORK_CLEARTEXT, for the LogonUserEx Win32 API function that results in a network logon but preserves the credentials in cache for delegation purposes. This gives the impersonation account a network logon security context rather than a local logon without sacrificing the ability to delegate.

By default the IUSR_MachineName anonymous impersonation account is granted the "Log on locally" right and is made a part of the Guest group. This means that access restrictions imposed on guest users who are allowed to establish interactive logins on the IIS box are the effective default restrictions for IIS while it processes anonymous requests.

However, there really is no such thing as a guest who you allow to logon anonymously through an interactive shell session. This type of interactive login is a figment of the imagination, and you can be certain that programmers don't consider it as a real possibility when writing application code and Windows platform APIs and features. This makes the IUSR_MachineName a little more dangerous than it would be solely as a network logon-privileged account. Because it is possible to ensure that IIS only ever authenticates to Windows with a network logon on behalf of the anonymous impersonation account, you should remove the "Log on locally" right from the anonymous impersonation account and replace it with "Access this computer from the network" instead. As long as you allow IIS 4 to control the password for the anonymous impersonation account the "Log on locally" right is not necessary to conduct anonymous impersonation.

A metabase property exists to clarify explicitly for IIS the way in which logons should be obtained during authentication events including Anonymous Authentication performed automatically by IIS on behalf of anonymous users. The metabase property is named LogonMethod, and its possible settings are numeric values ranging from 0 to 3. A value of 0 indicates logons will be performed using log on locally mode, meaning that the flag value LOGON32_LOGON_INTERACTIVE is passed to the LogonUserEx API call. A value of 1 for LogonMethod indicates that the logon will occur in batch mode, with flag value LOGON32_LOGON_BATCH passed to LogonUserEx. Batch mode requires the impersonation account to have the "Log on as a batch job" right rather than "Log on locally". When LogonMethod is set to 2, the logon is a network rather than a local or batch job logon. Flag value LOGON32_LOGON_NETWORK is passed to LogonUserEx.

Finally, a LogonMethod of 3 results in the new type of network logon introduced with Windows 2000 where credentials are cached to enable delegation. The flag value LOGON32_LOGON_NETWORK_CLEARTEXT can be passed to LogonUserEx by IIS 5 or IIS 6 when LogonMethod is set to 3 but IIS 4 and Windows NT don't support this LogonMethod setting. The LogonMethod metabase property can be specified for every level of password-protected resource including files, and impacts the call to LogonUserEx performed by every authentication method that supplies plaintext credentials to IIS from the client in addition to Anonymous Authentication.

For the latest information available from Microsoft about authentication under Windows .NET Server/IIS 6 see the following pair of Knowledge Base Articles:

Q324274 HOW TO: Configure IIS Web Site Authentication in the Windows .NET Server Family

Q324276 HOW TO: Configure Internet Information Services Web Authentication in the Windows .NET Server Family

ASP.NET Forms Authentication

Forms Authentication is the predominant end-user authentication method in use today because it offers good security with complete user interface customization ability for the login page. It avoids the problematic multiple prompt malicious insertion attack where HTML or client-side script is injected by way of XSS, MITM, or site content tampering (defacement) that can easily trick users into revealing Basic Authentication credentials by carelessly responding to password prompts based on the Realm. Forms Authentication does not, however, avoid transmission and storage of full credentials, nor does it work flawlessly with browsers that disable all cookies for privacy reasons. It has also been the source of countless Web application security problems resulting from the common misconception that anyone who knows HTML and a server-side script language can build their own reliable Forms Authentication mechanism. Forms authentication relies on session tracking techniques using session cookies or URL-encoded session identifiers with two important differences. Unauthenticated sessions don't receive authentication credentials from the user, and Web applications that don't support authentication don't have anything to protect from unauthorized access.

Everything discussed previously in Chapters 5 and 6 concerning ASP.NET and hardening of session identifiers applies to any Forms Authentication mechanism. Vulnerabilities inherent to URL-encoding make it prone to session ID disclosure. For example, REFERER HTTP headers can be sent to third-party servers containing the entire URL with its encoded name/value pairs. Browser history cache will store the entire URL and make it available to users and scripts at various times. Favorites store complete URLs by design. Web crawlers can store hard-coded session ID values in databases used by search engines, resulting in many Web clients instructed to use the same URL-encoded identifier handed to them by the referring search engine. XSS vulnerabilities can capture URL-encoded name/value pairs with relative ease compared to XSS cookie capture exploits.

Simple visual observation over-the-shoulder in a practice known as shoulder surfing will leak URL name/value pairs to any direct observer. These and other risks create a nearly intractable problem with respect to end-users who refuse to allow session cookies for privacy reasons. Cookieless browsers, that is browsers that won't even allow non-persistent session cookies (where the cookie has no expiration date and thus is never written to hard disk) force Web applications to resort to URL-encoding of session ID which invariably results in less security for the user and the application. Users concerned about privacy often fail to understand the adverse impact on security that their cookieless browser policy creates for session-oriented and password-protected Web sites. One solution to this problem is thorough education of your Web site user community as to the safety and privacy of allowing session cookies. Or, to support cookieless browsers

without resorting to URL-encoded session identifiers, you can force failover to Basic Authentication. If URL-encoded session identifiers can't be avoided, ASP.NET supports this mode of operation also with Forms Authentication.

To avoid problems caused by aggressive or misbehaving (or even maliciously misconfigured) proxy servers that cache HTTP responses containing Set-Cookie headers, ASP.NET Forms Authentication only drops its session cookie in response to a FORM POST in which the end-user supplied authentication credentials. In addition, the HTTP Set-Cookie header generated by ASP.NET is always part of a 302 Found (Temporary Redirect) that is marked as uncacheable (Cache-Control: private) to further reduce the likelihood that any proxy server would ever cache the HTTP response and its Set-Cookie header. When SSL is used to encrypt all communication with the client, authentication is a simple matter of verifying credentials and then dropping a session cookie that is truly random and lengthy enough to resist brute force guessing. Even without special countermeasures to detect brute force attacks or cookie theft and deny access to the Web application from offending IP addresses the extra security precautions implemented by ASP.NET Forms Authentication protect against common authentication security flaws.

Knowledge Base Article Q263730 titled Site Server Users May Be Authenticated Under the Wrong Account tells a cautionary tale of bad Forms Authentication.

See also the HTTP cookie spec: RFC 2109 HTTP State Management Mechanism.

Anatomy of a Forms Authentication Ticket

Forms authentication tickets are created by serializing an instance of the `System.Web.Security.FormsAuthenticationTicket` class, appending the site-specific encryption key configured by `web.config`, and applying a Message Authentication Code (MAC) including a cryptographic hash. The resulting MAC is added to the serialized object instance and the combination is Base64 encoded to facilitate storage in a cookie or transfer as a URL-encoded QueryString parameter. The following steps describe the Forms Authentication Ticket creation process in more detail.

1. The `FormsAuthenticationTicket.IssueDate` is converted to a `time_t` data type
2. The property values in the `FormsAuthenticationTicket` object are serialized
3. The server's encryption key is used to produce a keyed hash of the serialized `FormsAuthenticationTicket` properties
4. The keyed hash value, which is to be used as a Message Authentication Code (MAC), is appended to the serialized object properties
5. The properties and the MAC are encrypted using the server's encryption key
6. The resulting ciphertext is Base64 encoded

Forms authentication tickets are validated in `FormsAuthenticationModule` by repeating the serialization of `FormsAuthenticationTicket` properties and recomputing the MAC in order to compare it with the MAC provided in the ticket. This process ensures that the cookie value isn't easily reproducible without knowledge of the secret encryption key configured by `web.config` for the ASP.NET application. It also ensures that new tickets can't be guessed through bit manipulation algorithms based on the value of a ticket that is known to be valid. By first validating the MAC of any Forms Authentication ticket supplied in a

client request ASP.NET is able to reduce the likelihood of DoS conditions created by an attacker who sends bogus tickets repeatedly in order to consume CPU time performing unnecessary credential lookups.

Dynamic Modification of Credentials in web.config

Forms authentication's default authentication store, the built-in XML node <credentials> inside the web.config XML configuration file, can be updated programmatically. ASP.NET is notified when web.config changes and automatically rereads the file so new user accounts are available to be used in authentication events right away. The following C# code shows how to use System.IO and System.Xml classes to add user accounts. A simple HTML form with an input named "userID" and an input named "password" is assumed as a user interface to POST to this code in order to supply the user credentials that it adds to web.config.

```
<%@ Page language="C#" %>
<%@ Import namespace="System.IO" %>
<%@ Import namespace="System.Xml" %>
<% FileStream fs;
fs = File.Open(Request.MapPath("Web.config"),
FileMode.Open,FileAccess.ReadWrite,FileShare.Read);
XmlDocument xd = new XmlDocument();
xd.Load(fs);
XmlElement xe;
xe = xd.CreateElement("user");
xe.SetAttribute("name",Request.Form["userID"]);
xe.SetAttribute("password",Request.Form["password"]);
XmlNode xn;
xn = xd["configuration"]["system.web"]["authentication"]
["forms"]["credentials"];
if(xn.SelectSingleNode("child::user[attribute::name="" + Request.Form["userID"] + ""]) == null)
{
xn.AppendChild(xe);
fs.Position = 0;
xd.Save(fs); }
fs.Close(); %>
```

The first step is to load web.config into an XmlDocument object. The Load method is used for this purpose. A FileStream is passed to Load positioned at the beginning of the web.config file opened for reading and locked for writing. XmlElement xe constructs the necessary <user name="" password="" /> syntax through the CreateElement and SetAttribute methods of XmlDocument and XmlElement, respectively. XmlNode is used to store a reference to the forms credentials node inside the web.config XML document and SelectSingleNode is called with the following XML Path Language (XPath) location path:

```
child::user[attribute::name="" + Request.Form["userID"] + ""]
```

(The XML Path Language official specification is at <http://www.w3.org/TR/xpath>)

The `child::user` identifier refers to a child node beneath `<credentials>` named `<user>` and the `[attribute::name=]` pattern is matched by a `<user>` node with a name attribute equal to the `userID` passed to the script by an HTML form. Provided that the call to `SelectSingleNode` returns null indicating that no user account exists currently in `web.config` with the specified user name, `AppendChild` is called to place the new `XmlElement xe` inside the `<credentials>` node. The `FileStream` position is returned to the beginning and the modified XML document is written back to `web.config` with a call to the `Save` method of the `XmlDocument` class. The `FileStream` is closed to prepare it for garbage collection and release the write lock placed on `web.config`.

The code shown is useful for administrative functions but not for deployment in a production application that provides self-service user signup because the sample has no mechanism for handling file access contention. While one user request opens the file `web.config` locked for writing other requests to the page encounter exceptions. To deploy this code in a production environment you can use a message queue with a new user account creation service. Dispatch a message as shown in Chapter 15 to a new user service in response to a user's HTML form submission after verifying with a user name reservation service that the requested user name is available. The new user account creation service will lock `web.config` and write in batches to manage contention for access to `web.config` by other processes and administrative users.

Protecting sensitive resources and data that are made available to authorized users through a Web application is the ultimate goal of authentication. Preventing unauthorized access to data is one element of this protection. Ensuring that only authorized users are allowed to make certain requests, such as to initiate a business transaction or give instructions for the handling or transfer of assets, is another aspect. Simple HTTP authentication methods or Forms Authentication combined with SSL encryption offer very good security for access control, but are generally inadequate security when it comes to proving the identity of the person or entity that makes a business request of some importance. For positive proof of identity, password-based authentication methods offer insufficient safety. Client certificates are the preferred way to establish real trust for reliable identity verification when mutual authentication is required in a Web application. Chapter 14 shows how to create and use certificates for this and other purposes. Passwords aren't the best way to do things, but they are the standard of practice for now because they're easy to set up and easy to use. They are also easy to intercept or crack, but that's a reasonable risk for many applications.

Chapter 13: Trojans and Stealth Rootkits

Defending against Trojans is very different than defending against virus infections, worms and similar threats. Consider this premise: a sufficiently-compromised system may lie about its health when it conducts self-diagnostic tests. A Trojan designed to conceal itself by compromising operating system code is generally referred to as a stealth rootkit. A Trojan designed to actively evade detection with antivirus software or other tools by bouncing from RAM to hard disk to RAM and back again in various forms, hiding in some inconspicuous data through steganography at one moment and then piggybacking on an event triggered by a user or another program to decode itself on-demand is generally referred to as science fiction. But it isn't hard to design such a super-Trojan, at least as vaporware. One obvious evasive trigger would be for the Trojan to detect the launch of scanning software, write itself temporarily to hard disk, schedule a new task via Task Scheduler service, then purge itself from RAM just in time to avoid detection by the initial RAM sweep of the scanner. Then, shortly after the scanner switches to the time-consuming task of the hard drive search, the scheduler wakes up the Trojan again which quickly erases itself from the hard disk. Perhaps the Trojan encrypts itself with a dynamic secret key such as the local computer name so that its bytes on disk can't be identified by any Trojan scanner.

Or perhaps the Trojan stays off the hard disk completely, except for the final moments of system shutdown and the first few moments of system startup. RAM sweeps might detect the Trojan, but not if it moves itself around by forcing other processes that have already been scanned to become the new temporary host. A box that exposes a remote-exploitable vulnerability as part of its trusted codebase could even be compromised by way of a Trojan that exits the box temporarily in outbound encrypted network streams, erasing itself completely from RAM and hard disk while scans and self-diagnostic tests occur only to return later for another round of attack. Sure, such a super-Trojan would have to know about security bugs in software and system APIs that would give it this type of nimble box- or process-hopping ability, it would have to know about the design of virus scanning software, and it would have to be something that nobody has ever seen before so it might only be useful once or twice before methods of detection are developed. But to make the assumption that the bad guys (or the good guys!) aren't smart enough to build such a thing, or to assume that there aren't enough security vulnerabilities in commercial software and system APIs to facilitate such elaborate evasive stealth super-Trojans would be to ignore the basic facts that programmable computers do whatever they are programmed to do and security vulnerabilities exist in virtually every software program. This give-and-take process of better malicious code and better detection software is often termed an information security arms race.

Identifying Trojans is more of an art than is virus detection. To detect Trojans that people have seen before and profiled forensically is no different than conducting virus scanning. Easier still to believe than the prospect of a super-Trojan is a simple stealth rootkit that hijacks your antivirus software. Some people always use more than one virus scanner for this very reason. Preventing your IIS box from compromise by known threats and previously-discovered vulnerabilities simply requires staying abreast of infosec news and patches. It's the unknown threats that prompt us to prepare in advance to defend

ourselves and respond to security incidents when they do occur. A skilled and motivated attacker will go through great pain and expense to place custom Trojan code on a target box. Antivirus software is incapable of detecting most malicious custom Trojans, although a few well-known infection methods like the various Run registry keys are often monitored generically for any changes by antivirus programs. A good Trojan will use a method to keep itself active that isn't already known and over-used by every script kiddie.

The effects of a Trojan are often more subtle than are the effects of a virus. Defending against the unplanned introduction of code to production systems is an art because it relies on more than just security tools it also relies on the human ability to perceive suspicious activity and circumstances. Computing style, that is the character of usage patterns that typify the way in which a computer system is handled, determines whether there is high risk of Trojans in the computing environment. Of secondary importance are vulnerabilities in services and open ports that might allow a Worm or a targeted penetration attempt to gain entry to a protected system initially to plant Trojan code.

Your IIS box can be protected against Trojans successfully only if you assume that it has already been compromised by an ingenious stealth rootkit. Guilty until proven innocent, but not yet convicted of wrongdoing or removed from service: this must be how you regard each IIS box that you program or manage. Every IIS box is always on the verge of being given the death penalty, anyway, either by you or by an attacker. The eagerness with which many administrators kill an IIS box when it misbehaves is frightening. Everything that an IIS box does that you didn't expect it to do is potential evidence of Trojan activity, evidence of software bugs or configuration problems, or it's evidence of your failure to understand some aspect of the function of IIS and the Windows platform. Regardless, it's not reasonable to throw out this important evidence, rebuild a server, and hope for the best next time because this practice proves that you wouldn't know if an IIS box was compromised in the first place. If you can't sit down in front of your IIS box and prove beyond a reasonable doubt that it is uncompromised and restore it to normal operations when something goes wrong, then all notion of trust of your IIS box is misplaced. Either you prove that a box can be trusted with specific evidence to back up this assessment or the box was never really under your control in the first place. The ongoing search for proof of Trojan activity requires just this sort of skill as a forensic investigator.

Blended Threat Characteristics

Any malicious program that doesn't self-replicate might be considered a Trojan, depending on what it does and who considers it. Classic Trojans give an attacker a means of gaining more elaborate entry to the compromised box at a later time, but remote access need not be a feature of all Trojans. When a program self-replicates it's either called a virus, implying that some user action is leveraged as a means of infection, reinfection, and transmission, or it's called a worm, implying a completely automated replication vector targeting a common vulnerability in other systems that are likewise susceptible to successive rounds of infection. A buffer overflow exploit in an ISAPI can be used by a worm to invade every IIS box that exposes that ISAPI. Hybrid attacks combine properties of both viruses and Trojans, with replication vectors normally associated with worms.

Concept Worms and Viruses

The first version of Code Red was a concept worm, demonstrating the potential to exploit an ISAPI vulnerability automatically from worm code but designed to do nothing malicious to infected servers other than use some CPU cycles to attempt to penetrate other servers. In this way Code Red was similar to the Robert Morris worm. Simply by worming its way through the network to reach nearly every vulnerable host in a short amount of time, such worms create DoS conditions by consuming CPU cycles and network bandwidth. Damages of such concept worms are usually limited to unscheduled paid time off for thousands or millions of people who can't get work done until the malicious code is purged plus the cost of system administrators' time to purge foreign code from infected boxes and harden them against repeat infections.

Concept worms and viruses are designed only as proof-of-concept code and are not designed to cause intentional destructive harm. The worst-case scenario outcome of infection with a concept virus or worm is an inexperienced administrator who misdiagnoses the problem and concludes that the box has to be retooled from scratch starting with reformatting its hard drives. Naturally the novice administrator (or do-it-yourself power-user) will fail to perform a proper system backup and important irreplaceable data is lost as a result of the attempt to restore the server to its previous uncompromised state. The urge to reformat hard drives and start over again from scratch when an IIS box becomes compromised is understandable, and retooling is a common remedy to any malicious code infection.

Zero-Day Exploits and Custom Trojan Code

When a brand-new worm, virus, Trojan, or attack method is first used to target victims' boxes the attack technique is referred to as a zero-day exploit. An attack method that is zero days old and actually works against systems in the wild is nearly impossible to defend against using conventional off-the-shelf software. Even when a commercial intrusion detection system (IDS) is deployed to protect a box against known exploits and known attack methods, zero-day exploits are often able to penetrate vulnerable systems anyway because the exploit takes advantage of something unanticipated by the makers of the IDS.

One of the reasons commonly cited for the creation of honeypots is to lure attackers into revealing zero-day exploit techniques against worthless computers so that valuable systems can be protected before the zero-day exploit hits. In reality it can take many days to analyze the forensic evidence gathered by a compromised honeypot. Meanwhile production systems that are also vulnerable to attack might also be compromised. This interim period between exploit development and definitive isolation, characterization, and countermeasure creation is the main reason to be concerned with the Trojan threat.

Code Red II was more harmful because it had a malicious purpose other than automatic self-replication. Code Red II used a similar attack method as Code Red but carried with it a Trojan payload in addition to the worm code. The Code Red II Trojan replaced explorer.exe with malicious code that would keep itself installed and guarantee that remote access was available to an attacker by way of file shares. Code Red II also placed a copy of CMD.EXE named root.exe in the scripts directory of each infected IIS box so that root.exe is accessible remotely in the future. Existence of any new worm is often detected first by monitoring of network traffic and log files. It doesn't take long after

first isolation of worm code for information to circulate that a new threat exists, but deploying immediate barriers to prevent an IIS box from being compromised can be difficult because worms often replicate faster than forensic analysts can reverse engineer and profile them adequately. A sufficiently prepared attacker will design and release many variants of the same malicious code simultaneously. You can't rely on industry warnings and infosec mailing lists to give you sufficient advance warning, even if you have a way to quickly update or reconfigure every IIS box under your control: zero-day exploit code will always have the potential to get to your equipment before word of its existence can reach you.

Spyware and Other Trojans Associated with Clients

Many Trojans are only a threat to client computers under normal circumstances. A Trojan designed to spy on the Web browsing activity of an end-user should be irrelevant on an IIS box because you purposefully avoid ever browsing any Web content while logged into the box as an interactive user for administrative reasons. However, any type of spyware can be placed on an IIS box by an attacker, and you should always assume that every keypress and other action that you take while logged into the box interactively is being logged and possibly transmitted in real-time to a remote location. With this in mind, consider whether or not there are remote exploitable services or security holes exposed by your IIS box that an attacker could take advantage of with the help of spyware to capture the administrator account password. The best answer to this question is "No." so that credential capture is irrelevant to the security of your IIS box unless physical security is compromised too.

Vendor Relations and Infection Prevention

It's a sad fact that many vendors just don't take sufficient action to educate customers as to the threat represented by unpatched code that contains high-risk security vulnerabilities. Some vendors won't even build and release security patches until after malicious code appears that exploits a newly-discovered vulnerability. This happens for a variety of reasons, including prioritization failures during security alert triage where the vendor's security team misdiagnoses the scope and nature of a newly-reported vulnerability. Very smart people often disagree about very obscure technical details that may or may not be mitigating factors that reduce the threat level of a particular software bug. Nobody can disagree, however, that a significant new vulnerability exists when millions of computers become infected with a worm or virus because the fact of the outbreak of new malicious code is as indisputable as its proven method of attack. For this reason some white hat hackers make it a practice to build and release concept code. There may even be value for security of a nation's computer infrastructure for certain government agencies to cause strategic large-scale disruptions by way of concept worms and viruses where the concept code is incapable of causing any real harm. An entire nation of computers can thus be inoculated against real threats that stem from a particular vulnerability, making the concept code analogous to a vaccine that prepares a biological immune system to fight off infection or disease caused by exposure to similar contagions that pose a threat in the wild.

Is it unethical to write and release a concept worm at the request of your government's computer infrastructure protection agency? Probably. Unless somebody else has already

unleashed a harmful variant and yours infects hosts for the purpose of protecting them? Complicated questions that provoke emotional debate. A different way to approach this issue that many advocate instead is the introduction of new product liability laws for computer software vendors. Something along the lines of requiring all vendors to attempt to communicate security alerts to customers and provide instructions for securing vulnerable systems. The product liability would in principle end within days of the release of the security alert, making it impossible for any person or company harmed by the vulnerability to claim damages in a civil lawsuit after a short window of time expires during which customers must apply security patches or follow the prescribed procedure in order to avoid invalidating an implied warranty of safety. But this sort of thing calls into question the precise legal definitions of “attempt” and “communicate” and creates jobs for an army of attorneys.

This also leads directly to technical discussion of digital signatures. Microsoft relies on digital signatures exclusively as a means of authenticating trusted code that the company actually published and Microsoft provides tools to automate the download and installation of trusted code. Trust is always determined automatically by verification of Microsoft’s digital signature. Other vendors take a do-it-yourself approach by publishing a list of “authentic” hash codes for the program files they’ve compiled and distributed. You know you’ve got the vendor’s authentic executable bytes when you can verify its hash code, but without digital signatures how do you know you have the right hash code? This catch-22 is believed to be resolved best by the use of digital signatures only, but in fact the only solution to the problem of trust verification for software lies in combining both automated digital signatures with human verification of the authentic hash codes that are known (to the person, rather than the computer) to correspond to a vendor’s authentic, trustworthy software.

Hotfix Digital Signatures and Authentic Code Verification

A digital signature is valuable for certifying authenticity of communications but only when those communications are evaluated by a human to give their content meaning and context. Digital signatures can also be used by automated systems to detect whether particular code or data are trustworthy, but only when you control the private key used to produce the signatures. There are few circumstances in which it is appropriate for an automated system to make trust determinations based on digital signatures produced by a third party. To override the human factor in the trust equation removes security context considerations like: “our company is currently under attack by insiders and a multitude of remote network nodes and other assailants therefore now is not the best time to allow the installation or execution of new software that appears to originate from our vendors.” When a vendor applies a digital signature to a program module and sends it to you, or when you retrieve such a program from the vendor’s Web site or a CD/DVD that you believe they sent you, it’s tempting to let automatic signature verification occur if this feature exists as part of a vendor’s system. Doing so is extremely dangerous, however, because you have no way to know with certainty that the vendor really signed the item in question. Before you decide to trust the authenticity of any digital signature you must evaluate all available information that might call into question the signature’s veracity.

For instance, suppose you receive a hotfix containing an updated inetinfo.exe module.

There are a number of things you can and should do before you allow the new inetinfo.exe to execute on your IIS box. First you compare the file size and other superficial properties of the new module file including its version information, which you can access easily through Windows Explorer by right-clicking the file and choosing Properties (see Figure 13-1), against the same information present currently on the version of the inetinfo.exe file that you intend to replace. There are a few things that you expect to see when you examine the new file's properties, and any violation of these expectations must cause suspicion. First of all, Microsoft often tells you what the new file version number is of the update they've provided by way of the hotfix. When this information is not provided, you simply assume that the file version number of the update will be larger than the previous version number. An attacker may attempt to trick you into installing an old version, or she may plant an old version inside what appears to be a new hotfix. Obviously it does you, or your automated system, more harm than good to verify Microsoft's digital signature on an older version of a file when you are attempting to install a newer version. When you trust automatic signature verification this sort of detail, the context of the action you are about to take, is missing from the trust determination algorithm.

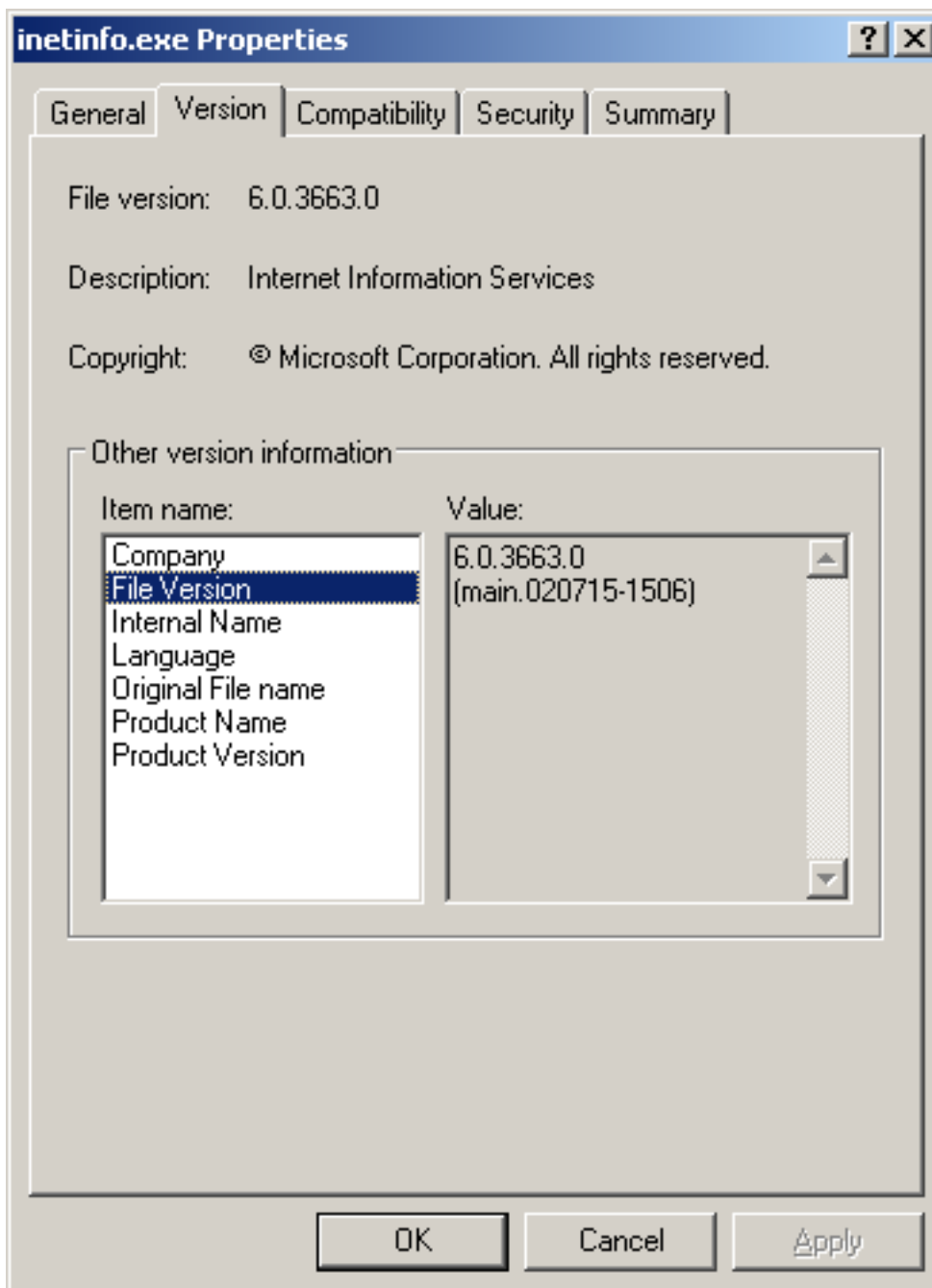


Figure 13-1: Inetinfo.exe Properties

A similar problem with automated trust systems based on digital signatures arises when downloadable code components like ActiveX controls are signed by vendors with valid Authenticode signatures and then shipped to users. It's very likely that the code as shipped and signed by the vendor contains security flaws, possibly even remote-exploitable stack buffer overflow bugs or other weaknesses that will allow an attacker to take control of a victim's computer by deploying the signed ActiveX control to purposefully execute authentic (yet vulnerable) vendor code that the attacker is prepared to exploit. If the victim's OS is designed to automatically trust the signature of a vendor who ships buggy code, the victim is unable to prevent the vulnerable code from executing even if there would otherwise be good reason to deny it that ability at the time of

signature verification. A vendor may sign and publish vulnerable code unintentionally, but this common blunder still gives attackers a new way to take control of computers that automatically trust all vendor code. In addition to comparing a program file's version number with the currently installed version of the same file you should make sure that the new version has a reasonable file size compared to the previous version. A file that grows or shrinks in size dramatically from one version to another deserves a good explanation. When you don't have a previous version against which to compare it's still important to evaluate each file to the best of your ability.

Windows File Protection security catalog (.CAT) files contain hash codes of trusted, authentic binaries and other non-executable files published by Microsoft and other vendors. Any file that hashes to any hash code found in any installed security catalog file is automatically trusted as an authentic file that has not been tampered with or otherwise compromised. Never mind the fact that malicious .CAT files can be installed and authentic .CAT files can be removed, or that there's no way to know, when given a .CAT file full of hashes, just what the code does that is being authorized by the .CAT file or what the code's potential vulnerabilities are as-built by the software vendor. There is also no way to know whether a program installer itself contains or is a Trojan. An installation program that supplies a signed security catalog containing authentic hashes of the files it installs may be conducting a seemingly-legitimate file installation merely as a cover for its true malicious purpose. Whenever you copy new code to your IIS box that requires an installation program to be run, you have a different problem of authentication to contend with that Microsoft's security catalog files do not address.

It may seem obvious, but you should also take careful note of the installation wizard prompts displayed on the screen when you install the hotfix. The hotfix filename is not part of the information that Microsoft hashes, therefore you can't assume that the filename is a legitimate indication of the content of the file. In many cases the only chance you will have to detect that you've been given a mislabeled hotfix that contains old (vulnerable) code is a Wizard splash screen like the one shown in Figure 13-2 for hotfix Q321599. Like any hotfix or other digitally signed executable file, the Q321599 hotfix installer can be renamed so that it appears to be a different hotfix entirely. If you aren't aware that this is possible then you may not give proper scrutiny to the Wizard user interface when it prompts you to approve hotfix installation. Assuming a hotfix is what you think it is based on its filename and the fact that it bears a valid digital signature is an invalid assumption.



Figure 13-2: Q321599 Installation Wizard

Further complicating your effort to ascertain the trustworthiness of each executable bearing a valid Microsoft signature is the fact that the signature itself tells you nothing about the item that has been signed. You have to trust that Microsoft's programmers always create a friendly user interface that halts program execution or installation until a human user gets a chance to review the content of the user interface to verify that the program is what you expect it to be. It is not valid to trust something just because it bears a valid digital signature. Figure 13-3 shows how signature review alone is inadequate analysis upon which to decide whether or not it's safe to execute a given file. Aside from the timestamp, which is a clue that something is amiss only if you have an authoritative source of information that suggests the date and time that Microsoft signed the authentic file is different from the timestamp you observe on the file, there is nothing in the digital signature data itself that confirms that the signed file contains hotfix Q321599.

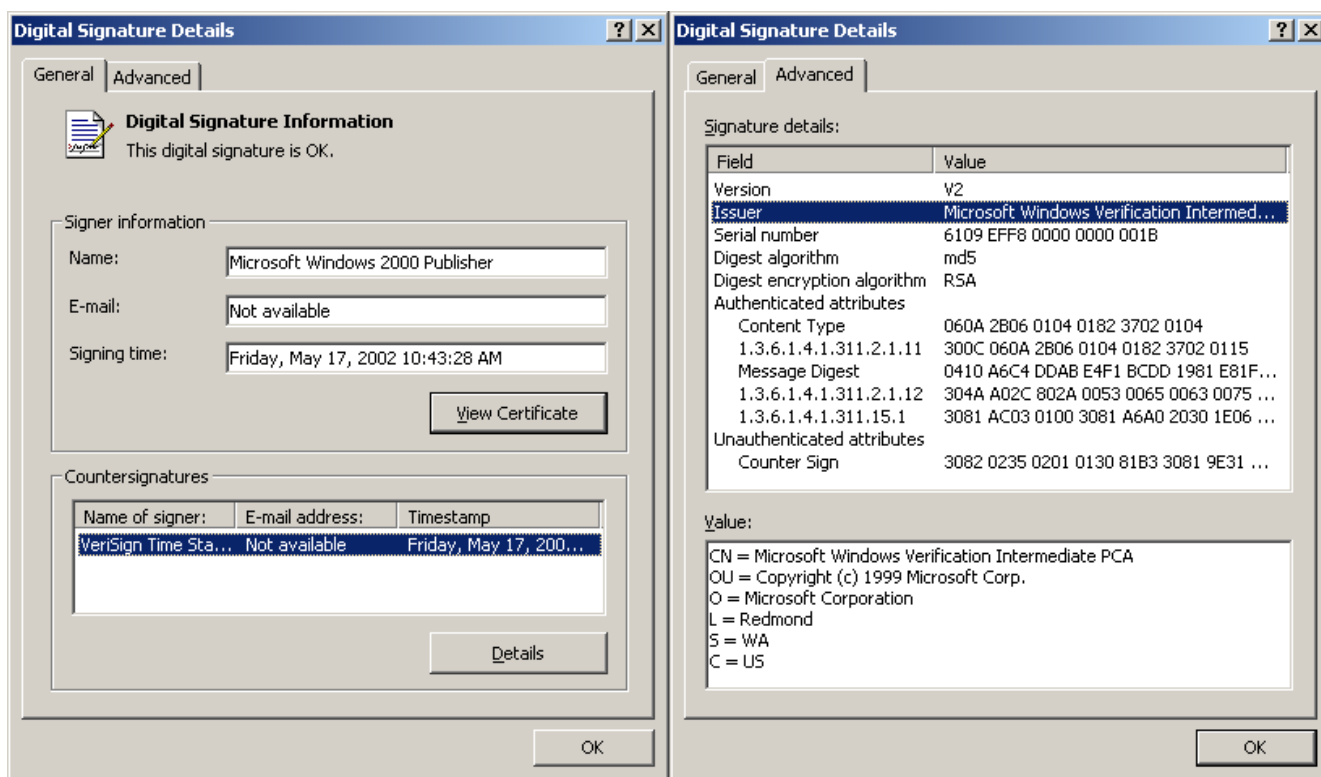


Figure 13-3: Q321599 Self-Extracting Cabinet File Signature

When a digital signature is attached to a portable executable it is placed in the Certificates data directory entry in the executable's IMAGE_DATA_DIRECTORY table located at the end of its PE header IMAGE_OPTIONAL_HEADER structure. This header is not truly an optional header but rather is included whenever an executable module is built and linked as a PE file for Windows NT rather than as a NE file for 16-bit Windows, or an executable for MSDOS or other binary executable format. As with any file that contains a digital signature, to verify the signature a hash of the signed data is computed and compared against the hash embedded in the signature data. The file itself is therefore different than the signed data because of the addition of a digital signature, which adds data to the file. This is one of the reasons that digital signatures are often simply trusted: the hash of the file changes when a digital signature is added, and only a program that knows how to extract the contents of the digital signature and apply the right hash algorithm to digest just the portion of the signed file that contains signed data is able to verify that the hash embedded in the digital signature matches the hash of the data that has been signed. Nothing in the digital signature itself as shown in Figure 13-3 is useful to you, the signature is useful only to software that is designed to verify it. The trouble with this situation is that it requires software to self-verify while installed on a box that might be compromised.

Much of the Microsoft code that bears a valid digital signature is designed to perform some task automatically when it is executed. There is no guarantee that a hotfix like Q321599 will present you with a user interface prompt that you can use as confirmation of the code's purpose before you allow it to proceed with installation. If you are presented with no user interface, and you can't determine the program's original filename by examining either its properties or its digital signature, then there is substantial risk that the code is not what it appears to be and it should not be executed on a production IIS box until it is

first validated on a stand-alone test box. In most cases you will have to conduct a trial installation of any new code in order to analyze it further such as through viewing the properties of individual files in order to compare the version number and timestamp against older versions of the same files installed currently on your production box. Most hotfixes accept a few command line parameters as well, and the typical `-?` parameter will usually reveal a list of available command line switches like the ones shown in Figure 13-4 supported by Q321599. Note that when the `-q` parameter is specified there will be no user interface, and `-u` sets unattended installation mode where user interface windows may appear but they will not prompt for user input. The hotfix Q article number is displayed in the title bar of the setup parameters dialog window, and the content of this title bar can't be modified without altering the hash code of the executable file as computed by Windows during digital signature verification.

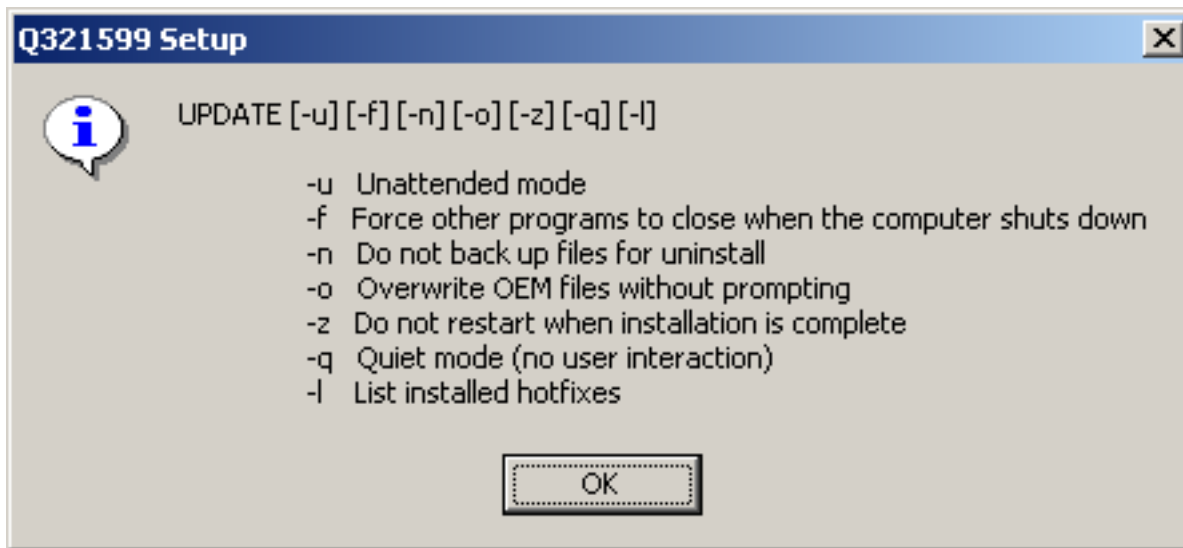


Figure 13-4: Q321599 Hotfix Setup Options

Using built-in signature verification features of Windows and then executing each program to find out whether or not it is the program that you expect it to be are critically-important preventative steps to take prior to deploying any digitally signed code to your IIS box.

Perform these steps on a test system that is kept physically secured and disconnected from any network or communications device. Physical separation between a test box and the network or other vulnerable access point, sometimes referred to as an air gap, is especially important for the system that you use to conduct pre-installation testing and verification of code authenticity. Better still, perform code authenticity verification on an air gapped box that has a different microprocessor architecture than the one for which the executables are built. When the code you need to verify is compiled for x86 you can most safely verify that code on a test box that runs something else entirely, such as on a Windows CE device. Ideally you would not have to execute the signed code in order to find out what it does, even when you think it probably contains setup options like those shown in Figure 13-4 that might make it unnecessary to do a full install of the code in order to verify its purpose. If the install program doesn't support any command-line parameters then any you supply will be ignored. After you verify its digital signature, the only way to verify that a program is what you think it is without executing it to see what it does is to compute a hash code of the entire program file and compare the hash against

a list of authentic hashes that your vendor published for the purpose of helping you to identify and authenticate trustworthy code that is safe to deploy.

When a vendor publishes an authoritative list of authentic hash codes of every file to which the vendor has applied its digital signature the real usefulness of digital signatures for Trojan prevention increases. This is just one more reason that digital signatures are more important for people than they are for computers. Even without digital signatures, you could use a trusted list of authentic hash codes published by a vendor to certify the authenticity of each program file distributed by the vendor. The practice of publishing authentic hash codes has been in widespread use for years by many vendors, and the trend to using automated digital signature verification instead as its next-generation replacement has severe unintended consequences that are just plain bad for the practice of information security. Whether or not Microsoft ever adds the publication of official “authentic hashes” to its information security procedures to protect customers from the threat of misplaced trust possible with the automated use of digital signatures alone, a little preparation beforehand enables you to acquire a trustworthy list of authentic hashes using simple forensic tools you can build yourself.

CryptCATAdminCalcHashFromFileHandle and SipHashData

The Microsoft Crypto API defines a system of Subject Interface Packages (SIPs) where objects that are manipulated by cryptographic operations, such as PE files, are associated with code that knows how to parse or handle them. This enables Crypto API functions that implement hash digest routines like Secure Hash Algorithm (SHA-1) to be applied to files with complex formats that can include digital signatures as ancillary data. Whether or not there is a digital signature included in the file data, the optional header or other placeholder for a signature can be ignored while only the relevant data is supplied as binary input to digest algorithms. Without SIP providers, every file format would be forced to implement digital signature attachments in precisely the same way or not at all.

Attaching a digital signature to a file is just one way to manage digital signatures, the security catalog file (.CAT) mechanism is a different approach that accomplishes the same objective but separates signatures from the signed files themselves. The problem with .CAT files is that they don't contain enough information to associate individual files with individual SIP-style hashes.

One of the most valuable abilities afforded to you by virtue of your possession of software hashes that are supposedly the authentic hashes of the software published by your vendor is the potential to compare authentic hashes with a peer. Insufficient information about the files being certified as authentic by a given .CAT file makes peer comparison more difficult. Windows File Protection utilities SFC.EXE and SIGVERIF.EXE will automatically compute the SIP hash of each protected file and verify that the hash is present in one of the signed .CAT files. By reviewing the log file produced by SIGVERIF.EXE you can even determine which .CAT file contains the SIP hash considered by Windows File Protection to be the authentic hash. But actually viewing the hash yourself requires a bit of custom code. You can choose to trust the validity of hashes of Microsoft code based on automated hash verification performed by SIGVERIF.EXE but the only way to know that your IIS box is running the same code as one of its peers is to perform a crosscheck by comparing hashes yourself. The following

C++ source shows how the Crypto API security catalog admin function CryptCATAdminCalcHashFromFileHandle can be called explicitly for a given file, mimicking the hash computation behavior of SIGVERIF.EXE. With the output of CryptCATAdminCalcHashFromFileHandle for each file you can do direct comparisons between file hashes on different IIS boxes and at the same time verify the consistency of the SIP providers that are responsible for WFP hashing.

```
#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <tchar.h>
#include <windows.h>
#include <malloc.h>
int _tmain(int argc, _TCHAR* argv[]) {
    char * filename = NULL;
    typedef BOOL (WINAPI * PFN_HFFH)(IN HANDLE hFile,
    IN OUT DWORD *pcbHash, OUT OPTIONAL BYTE *pbHash,
    IN DWORD dwFlags);
    HMODULE hm = LoadLibrary("mscat32.dll");
    PFN_HFFH f = (PFN_HFFH)GetProcAddress(
    hm,"CryptCATAdminCalcHashFromFileHandle");
    if(argc == 2) { filename = argv[1];
    if(strlen(filename) <= MAX_PATH) {
        HANDLE h = CreateFile(filename,
        GENERIC_READ|GENERIC_WRITE|GENERIC_EXECUTE,
        FILE_SHARE_READ|FILE_SHARE_WRITE,NULL,
        OPEN_EXISTING,NULL,NULL);
        DWORD hashlen = 0;
        BYTE * hash = NULL;
        if(f(h,&hashlen,hash,NULL)) {
            hash = (BYTE *)malloc(hashlen);
            if(f(h,&hashlen,hash,NULL)) {
                for(DWORD a = 0; a < hashlen;a++) {
                    printf("%hX ",hash[a]); }
                CloseHandle(h);
                free(hash); }}
        else { DWORD err = GetLastError(); }}}
    FreeLibrary(hm);
    return 0; }
```

CryptCATAdminCalcHashFromFileHandle can only be called by loading its DLL at runtime with LoadLibrary and obtaining its function pointer through the use of GetProcAddress. This is due to the fact that the import library for the DLL that contains this API function doesn't include it as one of the explicit public function exports. Under Windows 2000 the DLL to load is mscat32.dll while under Windows XP/.NET wintrust.dll is the preferred DLL although mscat32.dll still contains the CryptCATAdminCalcHashFromFileHandle function for backwards compatibility. The first call to CryptCATAdminCalcHashFromFileHandle in the code shown passes a null hash buffer pointer, which results in DWORD hashlen being set to the length of the buffer required to receive the entire hash. The malloc function is then used to allocate memory from the heap for the hash buffer and a second

call is made to the API function. This second call results in population of the hash buffer with the output of the SipHashData function. Figure 13-5 shows the Windows Debugger (WinDbg) at a breakpoint in program execution where the call stack includes the SipHashData call.

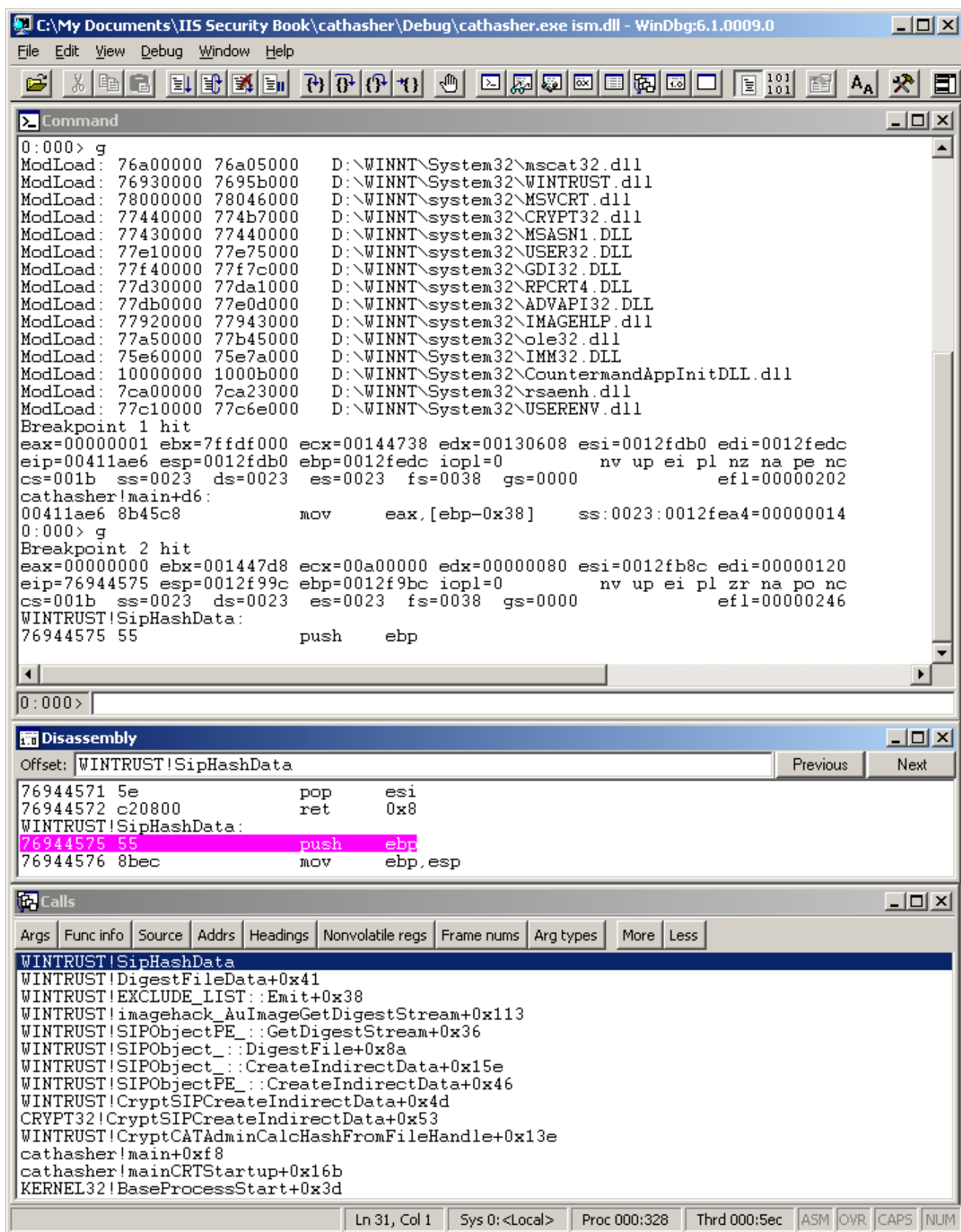


Figure 13-5: SipHashData is Called to Compute a PE Object's Hash Using a SIP Provider

Windows File Protection security catalog files contain conventional SHA-1 hashes of entire files when the files are not portable executables but still need to be protected. In this case the call to CryptCATAdminCalcHashFromFileHandle results in a hash that is identical to the SHA-1 hash you would compute with any SHA-1 hashing utility using any software on any platform that implements the SHA-1 algorithm. Figure 13-6 shows how similar the call stack is when a file is hashed using the CryptCATAdminCalcHashFromFileHandle API function and the file is other than a PE-formatted 32-bit Windows binary. SipHashData is still the call responsible for applying the right hash algorithm by calling into the appropriate hashing routine, but this time the SIP provider uses WINTRUST!SIPObjectFlat_ instead of the previous WINTRUST!SIPObjectPE_ object. Regardless, the stream supplied to SipHashData is the only data input that the hashing algorithm ultimately gets to drink. In this way there can be any number of prehashing stream manipulations and the SHA-1 hashing algorithm implementation available to the Crypto API doesn't worry about details.

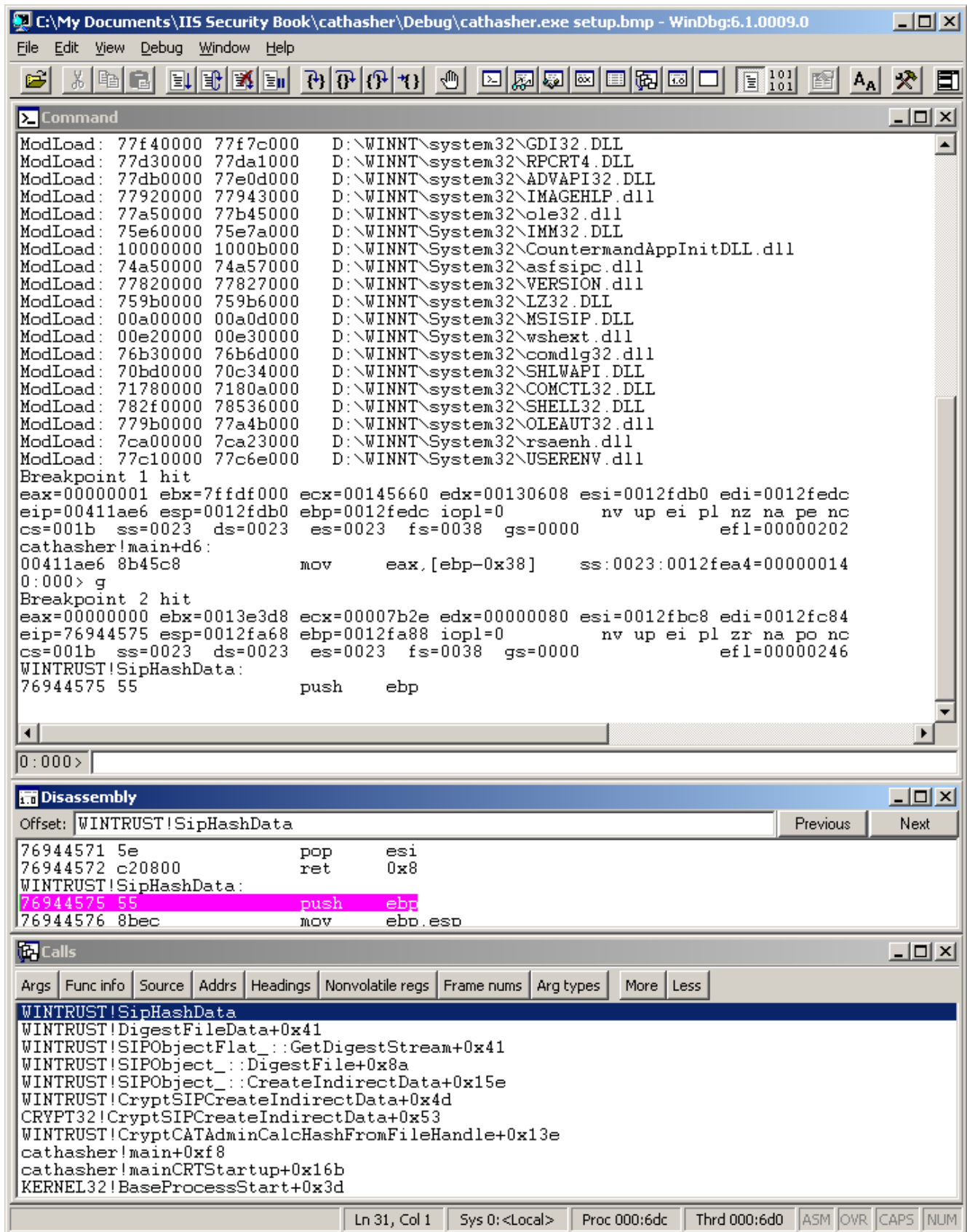


Figure 13-6: SipHashData is Called to Compute Hashes on Flat Objects Also

An authoritative list of authentic hashes, whether published by your vendor or computed manually and verified against a peer's hashes of the same files makes it possible to conduct code authenticity security audits of production systems without taking them offline.

Automated signature verification offers the best solution possible today for self-diagnostic protection against Trojans and tampering, but only a human can ascertain with 100% accuracy whether or not program code has changed since a system was first deployed. The process is simple, and not very time consuming. It can even be accomplished in many cases without system downtime. By computing your own hashes of the binaries that self-validate based on the vendor's digital signature and accepting the reasonable presumption that your system is not already compromised before it is put into live production mode, you create a checkpoint against which to verify each file later. By taking a forensic sample of each file you wish to validate from the running system through the use of a file copy utility located on a CD-ROM or other read-only media, you can verify that the hash codes of each file match your authoritative list of known good hash codes. For absolute confirmation you can take a production system out of service temporarily and mount its hard drives on a forensic workstation where drive images can be made for analysis so that there is no chance that malicious code present on the production box could possibly interfere with the forensic file copy. Hot-swap level 1 RAID drives (mirrors) are also useful for conducting forensic analysis without downtime, and hardware RAID systems further reduce the concern that malicious code might alter the mirror to make it appear as though the active primary drive has not been altered by tampering with the mirror's contents.

Don't forget that there are two different reasons for digital signatures on software published by your vendor. The first reason is simply to give you a better way to determine that the software your system runs is trustworthy and unmodified since its original creation by your vendor. The signature provides protection from the time the vendor applies their signature, not just from the time you install the vendor's software product. There are better ways to prevent modification after installation such as special hardware that enables an OS to operate from a write-protected drive. This achieves a superior level of runtime safety without the added complexity of digital signature verification, certificates, and trust management. The second, and most important reason for digital signatures is that they provide security that is nearly as good as physical read-only media with the benefits of authenticity verification and runtime reprogramming of your programmable computers.

SHA-1 File Hashing with .NET Framework Cryptography

Secure Hash Algorithm (SHA-1) is a hashing standard defined by the U.S. Federal Government's National Institute of Standards and Technology in publication number 180-1 dated April 17, 1995. Because SHA-1 is the hash algorithm used commonly for producing authentic hashes of software and other important files, it's useful to have access to a utility that will compute and display this type of hash on demand. The Microsoft .NET Framework includes two classes that implement SHA-1, one coded entirely as managed code, `SHA1Managed`, and one that wraps the native code Crypto API provider for SHA-1 used by `SipHashData`, `SHA1CryptoServiceProvider`. In the interest of using different code than that relied upon by Windows File Protection when

verifying SHA-1 hashes you can use the following C# code to produce SHA-1 hashes using only managed code. The hash codes produced for SIPObjectFlat type objects by SipHashData should always match the hashes computed with this code.

```
using System;
using System.Security.Cryptography;
using System.IO;
namespace sha1hasher {
class Class1 {
[STAThread]
static void Main(string[] args) {
HashAlgorithm sha = SHA1Managed.Create();
FileStream fs;
byte[] hash;
FileInfo[] fi = new DirectoryInfo(
Directory.GetCurrentDirectory()).GetFiles();
foreach(FileInfo f in fi) {
try { fs = f.Open(FileMode.Open);
hash = sha.ComputeHash(fs);
System.Console.WriteLine(
f.Name + ": " + BitConverter.ToString(hash));
fs.Close(); }
catch(Exception ex) {System.Console.WriteLine(ex);}
}}}
```

Because of the prehashing preprocessing of the file contents for SIPObjectPE type files performed prior to calling SipHashData, Windows File Protection .CAT files contain SHA-1 hashes that you can't verify with a simple SHA-1 hashing utility like the one shown here. Microsoft just didn't design Windows File Protection with the idea that any person would ever want to validate that the hashes of the software files they have installed on a box match the authentic hashes that are digitally signed and published by way of .CAT files. It's easier to let Windows File Protection do it all for you automatically, but certainly not more reliable nor safer than manual hash code validation. A combination of automated and manual verification of authentic hashes is the ideal solution, and Appendix A contains SHA-1 hashes of every file published by Microsoft for the base Windows 2000 product as of Service Pack 3, Windows XP as of Service Pack 1, and Windows .NET Server so that you can manually validate any or all of the files' authentic hashes. Any digitally signed file that you encounter beyond those listed in Appendix A should be hashed and logged for future reference. A comprehensive list of verifiable authentic hashes for your IIS box may prove to be very important to forensic analysis during an incident response.

The real purpose of going through all this trouble of examining the software you plan to install, verifying digital signatures, logging forensic information about each program file, and carefully analyzing installers before you execute them on your IIS box is not so much to prevent Trojan infections as it is to detect them reliably when they occur and recover from them without starting over from scratch. This gives you the luxury of taking a few risks here and there when there just isn't time to decompile, reverse engineer, and beat every bit of code to death on your forensic workstation – assuming you even have one

dedicated to this purpose – before you install it on your production boxes. Deadlines and such being what they are in the real world, you'll frequently download and install code without being 100% certain of its safety. In fact you should treat code that comes from any source, even off-the-shelf products purchased shrink-wrapped from a distributor, as suspect and conduct some amount of analysis before installing and executing it.

Relying on the Crypto API or .NET managed code implementations of encryption and hashing algorithms is a convenient shortcut for building your own forensic analysis tool, but doing so leaves something to be desired in terms of information security. A program that you compile yourself from source code that implements standard algorithms like SHA-1 without making any external API calls is a superior program because of its self-contained design and known source code content. It's not necessary to code your own implementation of the SHA-1 algorithm from scratch in order to achieve this level of code architecture certainty. But doing so is not terribly difficult. Federal Information Processing Standards Publication 180-1 defining the Secure Hash Standard SHA-1 can be found at the following URL:

<http://www.itl.nist.gov/fipspubs/fip180-1.htm>

Positive Proof of Signature Verification Failure

It's important to periodically verify the apparent integrity of your active SIP providers whether or not you decide to take the extra step of verifying authentic hashes on full files rather than trust SipHashData and CryptCATAdminCalcHashFromFileHandle to automatically compute less-than-full file hashes of your portable executable files and verify those hashes truthfully against the hashes contained in corresponding digital signatures. A production IIS box that appears to be uncompromised, that is able to compute the same SIP hashes as it previously computed with your forensic SIP hashing tool, can be trusted to a higher degree than an IIS box that doesn't have a verifiable checkpoint. However, a well-designed Trojan or stealth rootkit will know what the authentic SIP hashes are for each file published by Microsoft because the author of such malicious code has access to the same .CAT files as you do. It's very important to perform periodic tool calibration testing by hashing PE files that a malicious programmer has never seen before. An attacker can't possibly hard-code the correct SIP hashes of PE files they've never seen before, so drop a calibration test PE file that you build custom using a compiler. Before copying the calibration file to the production box, compute its SHA-1 hash and its default PE object SIP hash (its SHA-1 hash of portions of the PE file that aren't variable and optional) then use the same hashing tools on the production box.

Next, tamper with a PE or .CAT file published and digitally signed by Microsoft. A binary file editor like the one shown in Figure 13-7 is good for this purpose because it allows you to modify only a single byte of the binary file. This preserves the file format while modifying only inconsequential ASCII text data like the word "Microsoft" found in the file. You will alter the resulting hash code of the data while leaving intact its original digital signature and associated certificate chain. This makes it possible to attempt to verify the file's digital signature using Windows Explorer in spite of the fact that you've tampered with the file contents. The wrong changes to the file invalidate the file format, making it appear there is no signature.

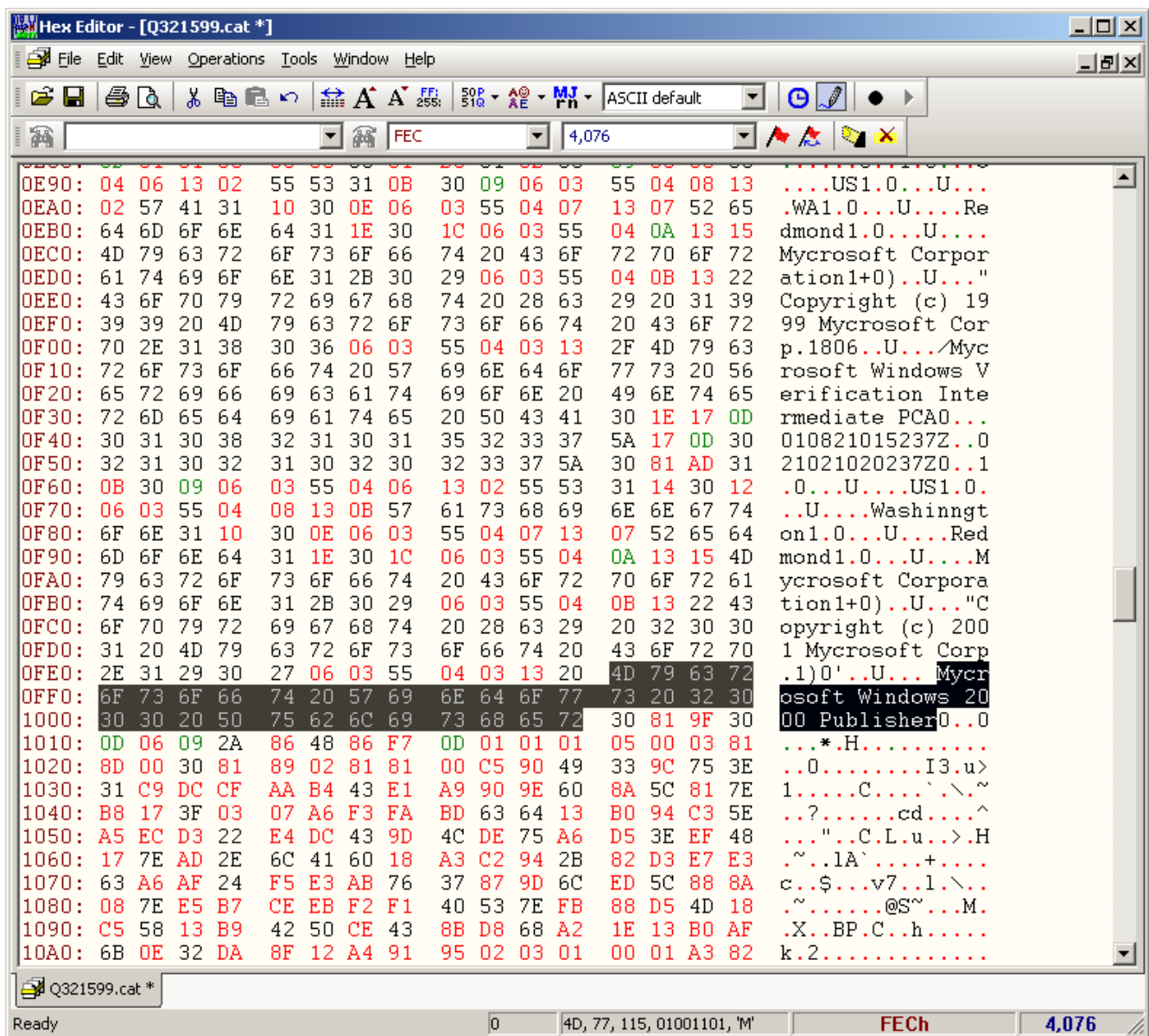


Figure 13-7: HexEdit Binary File Editor

If you rely on Windows Explorer to gain access to the default digital signature verification tool for Windows, your biggest concern must be that a Trojan may have altered this code so that it always affirms the validity of every digital signature. By creating a calibration test file that you know to be bad and attempting to validate its signature you can verify Windows' ability to discriminate between valid and invalid signatures in order to detect tampering. Figure 13-8 shows the signature verification failure messages that you should see when Windows is unable to verify the contents of a signed file as digested using the active SIP hash provider for the binary object file type and compared against the SIP hash encrypted with the signing authority's secret key then embedded in the signature data along with the certificate chain used to establish appearance of trust in the signature. This test isn't foolproof, but it is foolish to not perform it periodically. Believing everything you see on a computer screen is worse than believing everything you hear or read in print. As long as you mistrust your IIS box and understand that it can and will lie

to you if it's compromised by malicious code, you can be sure that it is still a reasonable risk to continue to use it.

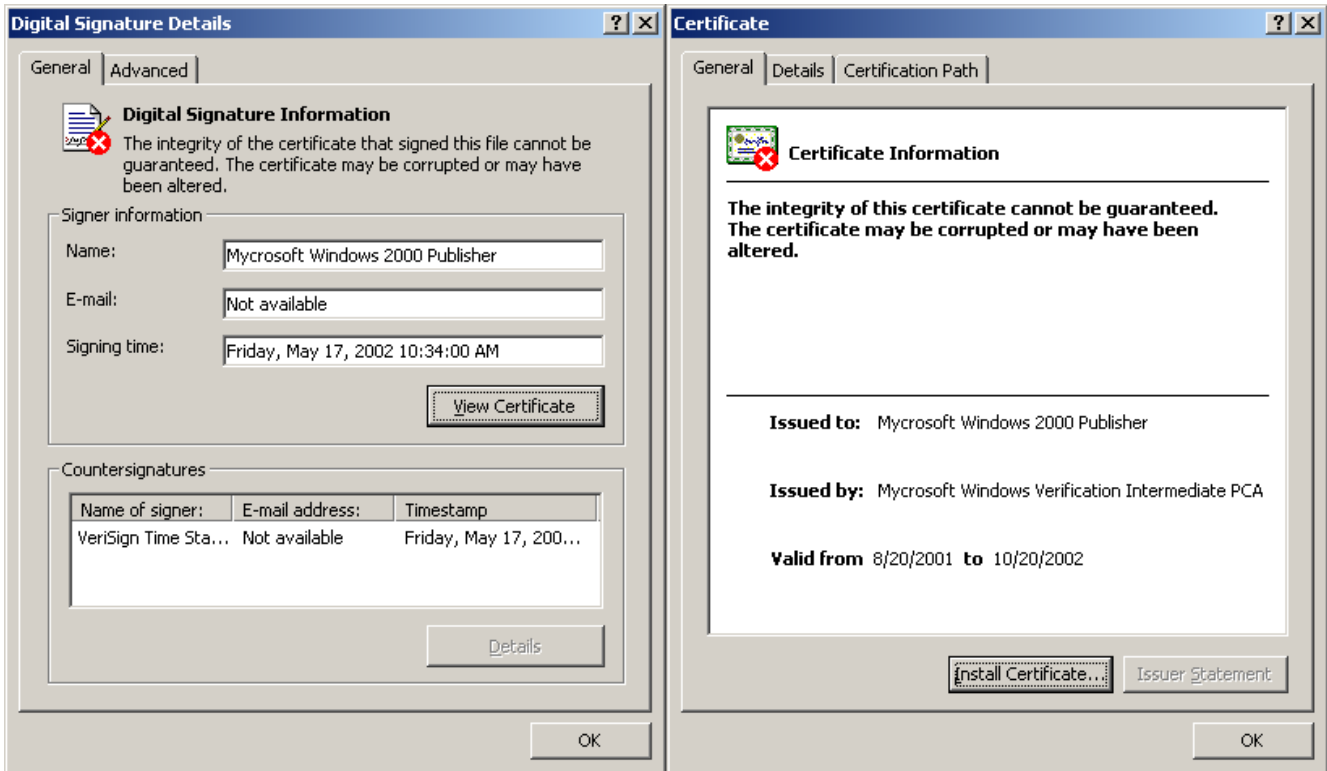


Figure 13-8: Digital signature validation failed

You may never be able to trust any programmable computer, least of all an IIS box due to its high-risk high-exposure function and role in your business, but it is still reasonable to continue to use programmable computer equipment that passes your safety calibration tests. If you don't bother to perform such tests or conduct forensic analysis of your trusted equipment then how can you justify continuing to believe it to be trustworthy? The essence of any Trojan is that it lulls you into a sense of security or complacency to take advantage of your laziness with matters of trust verification. Software vendors that try to deliver self-verifying systems based on programmable computers are often accused of selling snake oil because in reality there can be no such thing as trustworthy software unless and until there is trustworthy hardware.

Assume Your Vendor is Incompetent

It may seem unnecessarily paranoid to question the authenticity of a digital signature that validates using software that appears to implement signature validation routines correctly. Recall that it is computationally infeasible to discover a secret key through cryptanalysis of a signature, even though all the cryptanalyst has to do is try over and over again to sign the same bytes with randomly- or sequentially-selected potential keys until a matching signature is found. The number of possible keys is just too vast and the time required to discover the right key through such a brute force cryptologic attack is so immense, even with extremely powerful computers, that everyone concerned about the signature will be long dead by the time cryptanalysis succeeds. Besides, code signing certificates are intentionally given expiration dates so that a given signature will only

validate for a period of time before a new signature is required unless a new certificate is issued that recertifies the same public key. But consider what you are assuming your vendor does to protect their signature secret key from theft during the time that its certificate is valid. You are assuming that your vendor acts with extreme paranoia in order to protect against all possible methods of intercepting a signature key. This includes but is not limited to the use of sensitive equipment to detect the presence of listening devices, hidden video cameras, and other covert eavesdropping gear whenever the key is exposed, transported, used, generated initially, or discussed in the apparent safety of your vendor's office.

You are also assuming that the vendor will never use their secret key to sign something predictable, ever. If a vendor signs something that an attacker can predict the vendor will sign some day, then the attacker can be busy performing cryptanalysis in advance to build a database of possible signatures that result from every potential key. Therefore the vendor has to be extremely careful to ensure that the bytes being signed are unpredictable; that no attacker could have anticipated that the vendor would some day sign the bytes in question. More importantly, the vendor has to ensure that the bytes being signed are in fact the bytes they think they are signing, else an attacker could have replaced them with bytes of the attacker's choosing, making the resulting signature the same as the secret key from the perspective of the attacker since all they have to do is find the matching signature in their database of precomputed signatures to know the secret key used by the vendor.

Computer programs aren't entirely random. But then neither is any human language. It seems nearly impossible that an attacker would ever be able to predict the bit sequence of a compiled program long enough in advance to pose any threat to the security of digital signatures applied to program files. However, a vendor who is unaware of the risk of applying a digital signature to something like a bitmap image file that the vendor habitually includes in new releases of software, that the vendor has included in every release of its software for twenty years, may give away their new digital signature secret key when they ship a new-and-improved software release with a digital signature applied to that bitmap file for the first time. Cryptanalysts are smarter than vendors. This is precisely the type of thing a cryptanalyst would do, hoping that some day a target will slip up and give away the keys to the kingdom.

There must be an incentive for any attack. Script kiddies and others motivated by teenage angst (which, remember, lasts well into adulthood for computer geeks), boredom, and similarly innocuous impetus aren't likely to spend millions of dollars on equipment to mount sophisticated attacks unless a stock market bubble makes them suddenly rich without effort or reason. Oh, wait a minute... More importantly, even a well-funded and highly-motivated adversary won't bother with capturing a vendor's digital signature secret key unless there is substantial trust placed in that key already. A key that can do massive harm is a key worth the effort for a black hat to try to discover. This is precisely the type of key a vendor creates if digital signatures are used as sole protection for automated software distribution to millions of customers.

Windows XP/.NET Software Restriction Policies

There's no reason for your vendor to have the keys to your house or car. Why should any vendor have the keys to publish code to your computers? Microsoft Windows ships with a non-null list of trusted software publishers whose digital signatures can be automatically verified, and code is trusted by default even when it lacks a digital signature. To give you back control and final authority over all executable code that Windows allows to run, including interpreted scripts, batch files, and the like, a new feature was added to Windows XP and .NET Server known as Software Restriction Policies. Included as part of the Local Security Policy administrative tool is a simple UI for enabling and making changes to rules and rule enforcement settings for any specific executable or an entire class of executable content. Configuring restriction policies that implement a conservative, pessimistic view of executable content is one of the most important options available to protect against Trojans. While restriction policies do no good if you start out with a system that has been installed with a stealth rootkit present in the first place, as can happen if you acquire your copy of Windows from a source that itself has been compromised in advance with forged product from a careless or malicious distributor, an uncompromised system can in principle remain uncompromised more reliably with adequate countermand rules and code in place.

Software Restriction Policies are designed to apply by default to every type of executable content normally encountered in the wild. However, because Windows can be configured to support any number of additional executable content types and file formats, the executable file type list is configurable. Figure 13-9 shows the Designated File Types Properties window where you can review and edit the list of file types to which Software Restriction Policies are applicable. The default installation settings for Windows XP and Windows .NET Server leave it up to the administrator to explicitly enable enforcement of restriction rules for these file types.

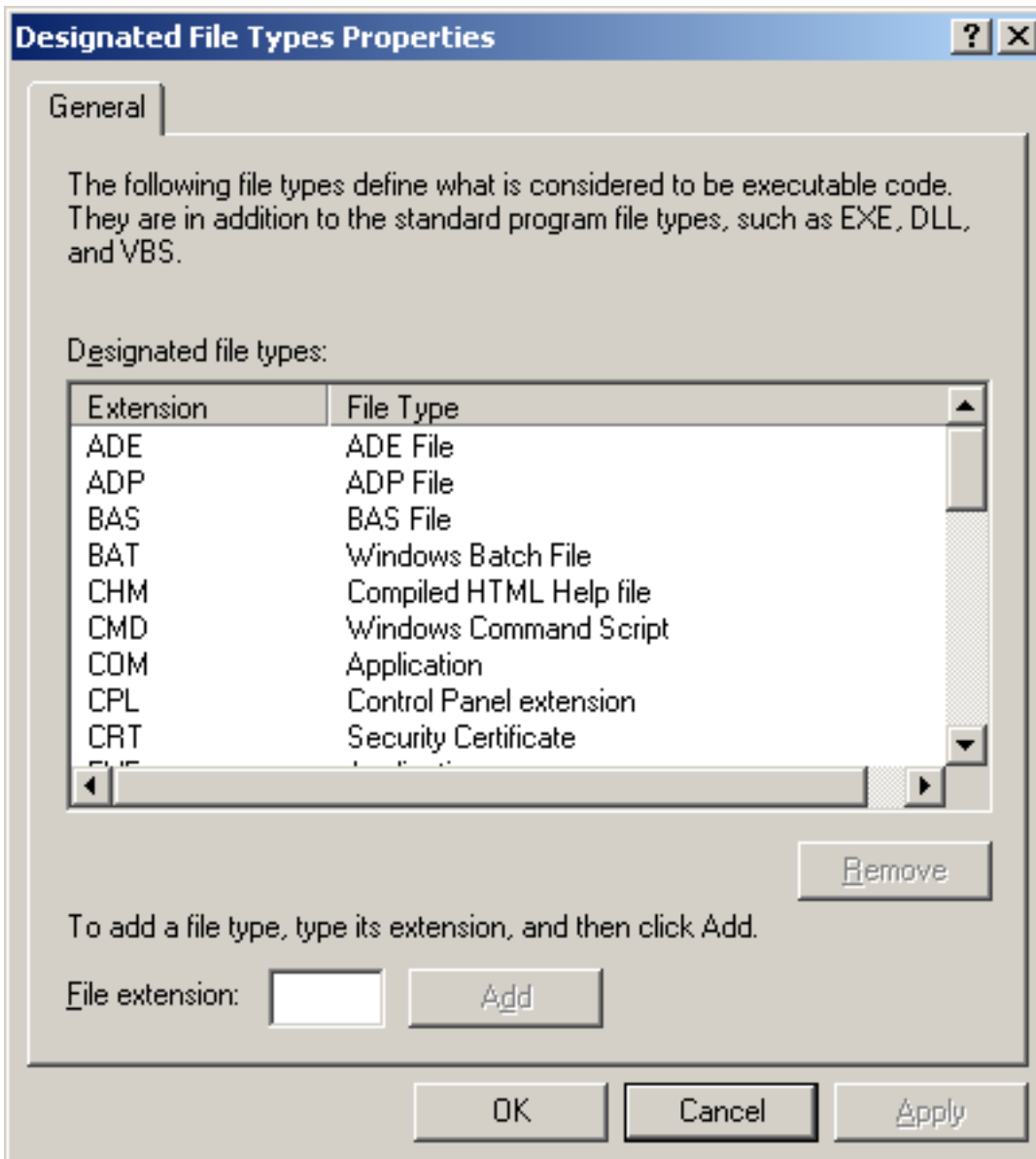


Figure 13-9: Software Restriction Policies Apply to Designated File Types

Figure 13-10 shows the configuration options available to fine-tune the way in which enforcement of restriction rules is accomplished for the box. In addition to choosing the level of protection desired in the Enforcement Properties window, All software files and All users or All software files except libraries and All users except local administrators, you can choose whether or not to allow end users to select trusted software publishers. Certificate revocation assurance properties are also configurable if Software Restriction Policies will be used to trust third parties' digital signatures where certificate revocation checks can be performed at run-time.

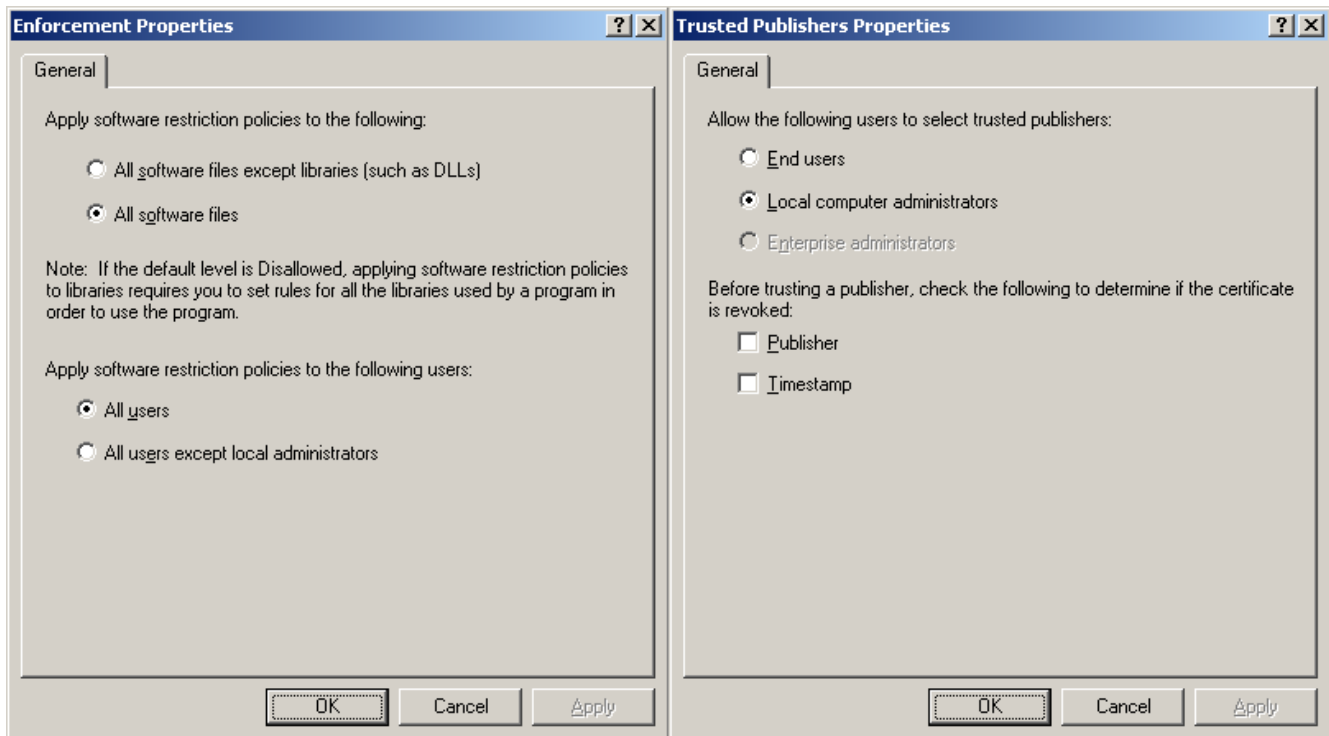


Figure 13-10: Configure Enforcement Properties and Trusted Publishers

There are many ways to configure rules for restriction policies, including selecting trusted software publisher certificates, allowing or disallowing executable files based on their full paths on the filesystem, using Internet zones, and based on hash codes. Unlike Windows File Protection, Software Restriction Policies will actually countermand the execution of software and scripts that do not meet the acceptance criteria defined by restriction policy rules. And any executable file may also be explicitly disallowed for meeting or failing to meet restriction criteria. It can take a long time to develop just the right restriction policies, and since this is a relatively new feature of Windows it still needs considerable improvement to realize its full potential, for example there is no way to profile its decision-making process to find out what it allows and what it disallows during a given time period, but benefits of adding countermand rules to local security policy are immediate and obvious.

Adding Yourself as The Sole Trusted Software Publisher

Preventing software from executing if it isn't authorized to do so is like juggling water. One of the complications of this entire proposition is how to determine, automatically, that particular code is authorized and other code is not. Hash codes and digital signatures are the rule of thumb when it comes to automating this type of decision-making, and digital signatures have one important benefit over hash codes alone: only a person in possession of a trusted secret key can produce the right sequence of bits (the digital signature) that will certify a particular hash code as authentic. Anyone can use a hashing utility to produce a valid hash code for a particular executable file. Although it is generally inappropriate to trust blindly and without further thought or analysis a file that arrives at a computer from elsewhere in the world (via CD/DVD disc, network connection, or other medium) there is one scenario in which it makes a lot of sense to allow automated trust of any file: when the file contains your own digital signature. By applying your own digital

signature to every executable file that you want your IIS box to trust, you can be certain that the only way for anyone to forge your signature is for them to steal your secret key. By carefully controlling access to your signed files, you can minimize the chances that any file bearing your digital signature will fall into the hands of a cryptanalyst, and without a specimen of your digital signature or your secret key the only attack possible against your IIS box requires physical or remote access to it, a condition that is relatively easy to detect. Chapter 14 explains the creation and use of certificates and public key/private key pairs. You must first create a certificate file and then add a new certificate rule as shown in Figure 13-11.

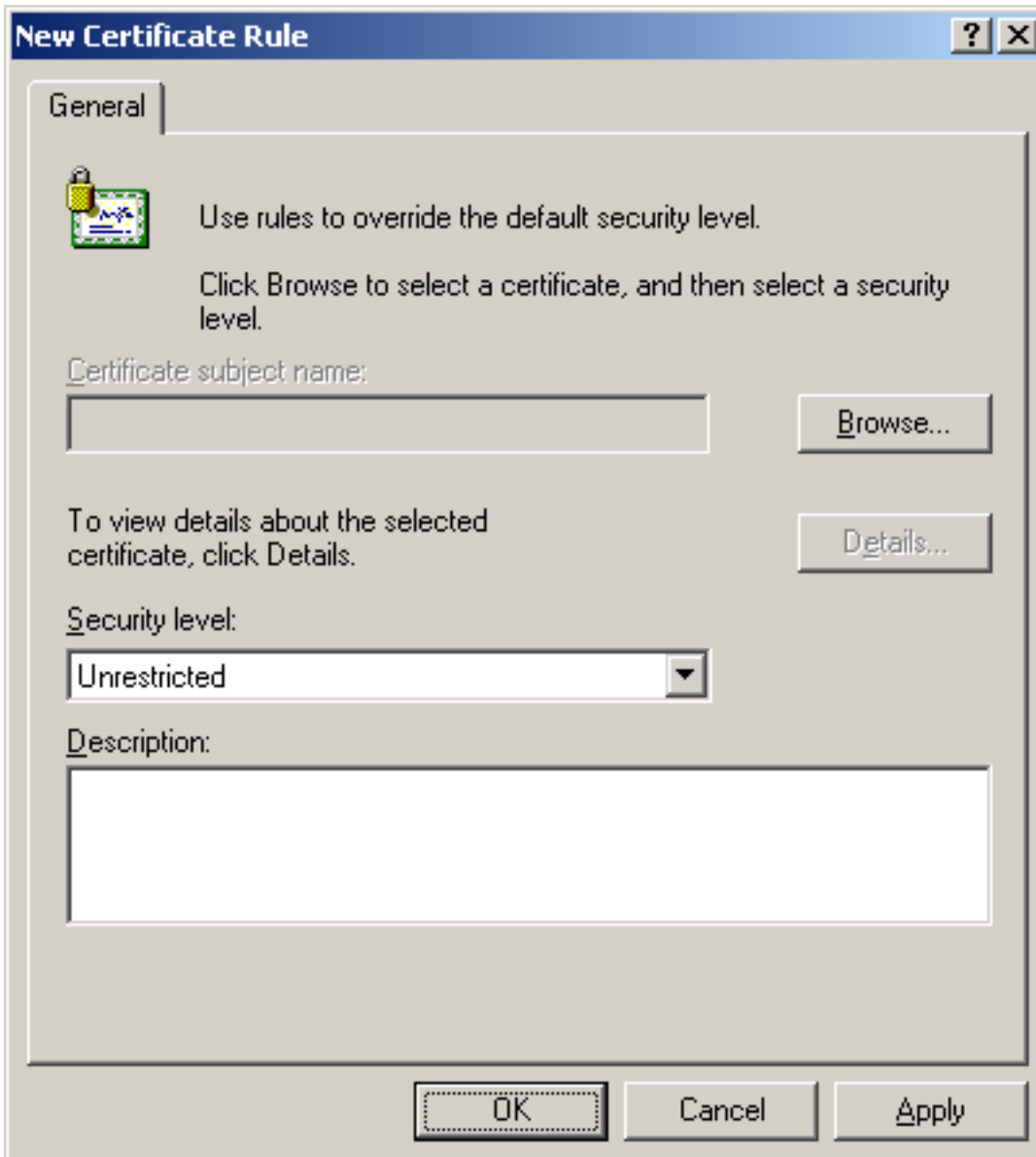


Figure 13-11: Software Publisher Certificate Unrestricted Security Level Rule

Trojans need not be compiled executable code, they can also be interpreted script files. The execution or interpretation of any file that is considered executable according to Software Restriction Policies' Designated File Types Properties can be controlled with a rule that explicitly verifies a supplied authentic hash code. When you can't apply your own digital

signature directly to a file because it doesn't conform to the PE file format, the only alternative to restrict execution of the file is explicit verification at runtime of the authentic hash code that you've determined corresponds to the trustworthy file. The combination of digitally signed PE files bearing your digital signature and preconfigured authentic hashes as restriction rules to allow files whose file format won't allow a signature to be attached offers an extremely good active protection against unknown threats.

Restricting Software Execution Based on Hash Codes

To add a rule that allows or disallows the execution of a file based on its hash code, you use the New Hash Rule window shown in Figure 13-12. Browse for the executable file and its hash and relevant attributes are automatically populated. Nothing prevents you from applying both a hash code rule and a certificate rule to every file. Remember that the hash code of a digitally signed PE file is different from its SIP hash, so you'll have to rehash any file that you resign after setting up a hash code rule. Because certificate rules take priority over hash code rules, setting up both is redundant. But this type of redundancy can be valuable when you have to revoke a certificate for some reason. After removing the trusted certificate, every signed file that was also specified in a hash rule will continue to have an Unrestricted security level while execution of any newly-introduced signed files will be restricted.

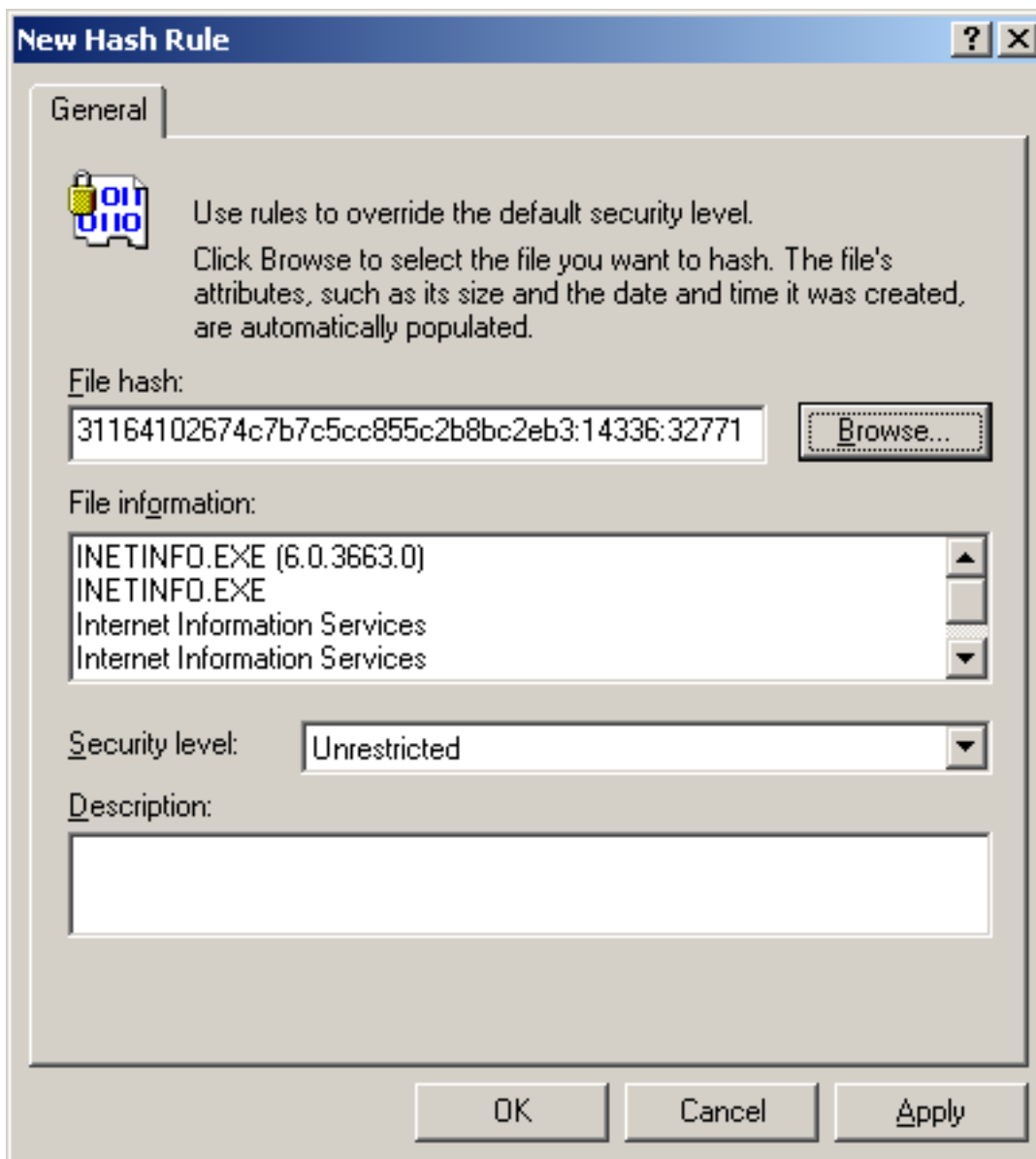


Figure 13-12: Inetinfo.exe File Hash Unrestricted Security Level Rule

In the future there will be enhancements to Software Restriction Policies that make management of rule lists easier, among other things. When you activate enforcement of restriction rules and set Disallowed as the default, without first adding rules that enable important OS files to execute, you can cause substantial problems for the operations of the Windows box. Much more could be written about techniques for utilizing Software Restriction Policies, and in particular there are important usage scenarios to consider in conjunction with Software Update Services for deployment of signed code files to many Windows boxes simultaneously throughout your organization's network, but there is no substitute for experimentation with SRP rules.

Trojans and stealth rootkits can tamper with data to alter perceptions and influence decisions or they can just lurk until an intruder invokes the Trojan code to gain remote access to your IIS box. For all the bad things Trojans might do, they're just software. As long as you have absolute control over the software that executes on your IIS box, including a

comprehensive list of the authentic hashes of that software, the steps required to protect against even the most nefarious stealth rootkit are no mystery.

Purging all Trojan code when a box is compromised can also be accomplished with relative ease when you know the difference between authentic code and everything else. Custom forensic tools and careful software deployment procedures that take into account the hidden dangers of automated software-based trust mechanisms are the most important additions you'll typically need to make to existing security policy with the goal of Trojan prevention. The WinTrust and Crypto APIs provide a starting point for implementing Trojan-proofing on a Windows box, provided you supplement automated analysis and trust determinations with manual forensic analysis to rule out the likelihood of stealth rootkit infections.

Chapter 14: Certificates for Encryption and Trust

The complex system of asymmetric cryptography combined with one-way hash code algorithms – message digest functions – and standards for distributing public keys to people and computers for the purpose of digital trust, collectively known as Public Key Infrastructure (PKI), is designed to do countless useful things. However, PKI should be used for only two reasons.

First, PKI enables you to apply bit sequences known as digital signatures to anything that you choose to trust or certify so that it is extremely unlikely that anyone other than you, the owner of a particular private key, could have applied the particular bit sequence mark. This technical feature enables another human being who has the right PKI software tools available on a trustworthy computer to ascertain that the mark appears to have come from you. There is value in this capability to convey knowledge to a third party through digital means that you probably possessed, in the past, a copy of the exact bits that are now in the possession of the third party. This gives the third party another reason to trust the communication if they believe it originated from you. Without this feature there is no way for other people to distinguish your authentic digital communication from obvious digital forgeries. However, it doesn't prevent forgeries; any cryptanalyst or thief can obtain a copy of your private key and forge digital signatures attributable to you.

Computers are not people. They have no common sense nor any concept of authentic vs. forged – they don't even communicate – they just move bits around according to the instructions that control them at any given time. Relying on other people's digital signatures for any purpose other than to help you, a human being, authenticate digital communication is an extremely risky misuse of PKI. Computers that are designed to automatically trust anything, be it data or code or instructions to take certain actions, based on automated verification of a digital signature belonging to an arbitrary third party, are not under your control they are under the control of whomever or whatever sends the data, code, or instructions. There may be limits on the scope of this control, or it may be absolute, depending on the design of the computer, its software and firmware, and depending on the content of the signed message. Communicating trust between people is a legitimate and proper use of PKI but automating the communication of trust between people and other people's computers is not an acceptable use of PKI. In a worst-case scenario, any anonymous remote attacker can cause arbitrary malicious code to execute by exploiting bugs in signature verification software in order to receive control of a box based on automatic trust of code that appears to contain a valid digital signature. This is a risk that it may be reasonable for you to take with your own signatures and your own computers but the apparent presence of other people's signatures should never be sufficient proof of the safety of code that will execute on computers that you own or control.

The second reason that PKI should be used is to certify the authenticity of security principals. A security principal in this context is just a bit sequence that is believed to be assigned to a distinct entity that is capable of originating, or receiving, digital communications. Because of the special properties inherent to these PKI bit sequences, they are superior to passwords and other types of digital authentication. In addition to an asymmetric key

pair (public/private) enabling the person or computer that controls a particular PKI security principal to demonstrate control of its associated private key on-demand, a system that uses PKI for authentication issues a digital signature of its own that binds a known public key to the physical entity that a corresponding private key is believed to represent. Such digital signatures are created by hashing and encrypting data, including the public key supplied by the owner of a key pair, in certain ways using the private key of a designated Certification Authority (CA). Some systems that use PKI for authentication operate as their own CA, while others trust third party CAs to apply their digital signatures to produce PKI security principals. The digital signature computed by a CA is attached to the public key that corresponds to an entity's private key and this combination is called a certificate. When a certificate is presented as authentication by an entity that claims to control it, two things occur. The digital signature applied to the certificate by the issuing CA is verified in the same way that any digital signature is verified. Next, the entity that supplied the certificate must prove it has possession of the private key that corresponds to the authenticated, digitally signed, public key. This proof occurs by encrypting a random message using the entity's public key as found in the certificate and then demanding that the entity decrypt the message in a cryptographic transformation that requires the corresponding private key.

The end result of all this is that the system performing authentication is able to determine that the entity claiming control of a particular PKI security principal in fact has possession of the one and only private key that corresponds to the certified public key. A private key is kept secret and is never revealed to anyone else including CAs that issue certificates to certify the related public key. Any system that must verify digital signatures on certificates issued by a CA has to be configured in advance with a copy of the CA's authentic public key. A CA certificate is therefore the root node in a chain of trust for subordinate certificates issued by the CA. Any CA can digitally sign any public key, including the public key that belongs to a subordinate CA, and issue a certificate. When one CA issues a certificate for the public key of another CA that itself will issue certificates a certificate chain is created. Any certificate except root CA certificates has a parent certificate and is thus part of a certificate chain. The only reason that certificate chains should ever be used is to enable human beings to verify a legitimate chain of trust supporting the authentication of a given public key.

Large-Scale Dangers of Certificate Chain Trust

Certificate chains are essential for the first use of PKI where a population of people need the ability to meet each other spontaneously and negotiate a shared concept of trust for their immediate and possibly future relations. The rationale being that "I will trust you because, in addition to having been validated by my common sense, you appear to be trusted by somebody that I trust." Certificate chains are disastrous in the second use of PKI because arbitrary valid certificate chains can easily be produced that will certify the authenticity of the same bits. Different people can have control of these mutually-contradictory but not mutually-exclusive certificate chains, and different CAs will usually be responsible for producing them in the first place. When authentication occurs based on certificate chains it results in a variable, open-ended trust model rather than the appropriate fixed, closed-ended trust model that common sense dictates. It also produces systems with subtle yet important security flaws.

Technically, the purpose of a certificate chain is to certify the authenticity of a single public key for a particular purpose or a list of purposes. Allowing any public key other than the one thus certified to be used in place of the one key that should be trusted, simply because an alternative public key is communicated by way of a certificate with a valid certificate chain, makes absolutely no sense. However, this is precisely the way that many PKI-based systems are designed and deployed by default. Many software programs that use PKI do so in a way that is open-ended with respect to trust chains offered by certificates, designed to make automated trust determinations based on arbitrary certificate chains so long as the digital signatures embedded in the certificate chain can be validated using trusted root and intermediate CA public keys. Arbitrary certificate chains that are trusted equally by PKI-based systems are a serious problem, and one that would not be a threat with more conservative security policies and procedures that force human intervention during the initial evaluation and trust determination of each public key. Every automated validation of a certificate chain not rooted at a CA that belongs to you is a dangerous condition just waiting to be exploited.

Choose A Public Key To Trust And Stick To It

Only humans should make trust determinations based on certificate chains whose CAs are controlled by a third-party. Public keys determined to be trustworthy based on review of a certificate chain should further be fixed, rather than open-ended, in any system that performs automatic trust determinations based on a validated certificate. It is inappropriate to allow a change in public key without human intervention yet this is precisely what arbitrary certificate chain trust enables. Any PKI-based software that allows open-ended automated trust determinations to be made based on the automated analysis of arbitrary certificate chains just asks for trouble. Automated use of a trusted public key may be appropriate for certain applications after a human has authorized a particular public key by installing a certificate. However, arbitrary public keys must never be trusted automatically for any purpose or else security is diminished and the value of PKI is destroyed. The design of PKI in Windows is seriously flawed in this respect because it fails to distinguish adequately between installing a certificate for the purpose of making it available during human-mediated trust determination decision-making and installing a certificate for the purpose of granting Windows the open-ended power to make automated trust determinations. Because of this design flaw you must avoid allowing Windows to get ahold of any certificate that it might be able to use as the basis of proving a trust chain to itself and instead force Windows to use only specific public keys that you know to be trustworthy for specific purposes.

There is no way to overstate the importance of preventing automated arbitrary certificate chain verification from being used as a basis of remote control over many computers. This practice, implemented by software developers including Microsoft, places human life at risk. A motivated criminal attacker who understands the power corresponding to the secret bit sequence of a root CA's private key if that private key can be used to take control of millions of computers automatically because those computers trust the root CA automatically, by design, will have no problem killing a few people to get at those bits.

This may seem like somebody else's problem, or a risk somebody else chooses to take, but consider that it is the application developer who causes this level of risk for a CA, not the CA itself. There are things a CA can do to defend its employees if they are conscious of

the type of risk to which they might be exposed when application developers misuse PKI, but the most important defense is to avoid misusing PKI and keep CAs informed of any bad practices you encounter. The more trust that is given to a particular CA the more risk there is that the CA itself will be attacked, and possibly using more than just computers.

Security Alerts Don't Resolve Certificate Chain Vulnerability

You're very optimistic if you think non-technical news media and industry forces opposed to full disclosure of information security vulnerabilities will clearly explain to you that the tragic workplace shooting that occurred today somewhere in the world happened at the offices of a root CA whose root CA certificate is trusted by millions of computers, and as a result everyone must act quickly to delete their certificate from every computer. By the time this happens it will be too late to educate everyone, by way of a 15-second news clip, that their computers are not really under their control but are instead under the control of a trusted third party who happens to now be dead. And whose root CA private key is now in the hands of another, untrusted, third party.

You may not have control over other people's poor security decisions, but you should learn from this scenario even if it never occurs: when you deploy a technical system that gives too much power to a single point of failure, and somebody else can kill you and steal the key to that single point of failure, you increase the likelihood of your own murder measurably. Further, if you build a computer system of extreme importance (e.g. military applications) that can be compromised and controlled remotely with possession of a particular bit sequence, you substantially increase the likelihood that the people who control that bit sequence will be killed for access to it. Amazingly, purveyors of PKI certificates knowingly take this risk and do nothing to educate developers about these issues even though substantial danger may emerge for the operators of CAs when information system assets of real value are protected with certificate chains they create for customers.

Managing Certificates and Certificate Stores

IIS rely on the certificate stores in Windows for validating digital signatures on certificate chains provided by the client and for constructing certificate chains sent to the client during client and server authentication events. Each certificate store contains a linked list of certificates that have been placed in that store by the user, an administrator, or by default. Each Windows user account has a personal MY store for user-specific certificates for things such as client authentication and personal digital signatures or Encrypting File System (EFS) use. The MY store can be in any of a number of physical storage locations, or a combination of several different locations including Active Directory, smart cards, Registry keys, or on the local filesystem. The ROOT certificate stores are the most important, as they establish the hierarchies of potential trust that an IIS box will consider to be valid at run-time. Each Windows user account can optionally have its own ROOT store that is different from the ROOT stores of other users or the local system default. Managing these various stores and establishing a secure policy for validating certificates and certificate chains is the basis of all security in PKI. As such, you should begin this ongoing process of trust management the way any good information security is achieved: deny all by default.

Removing Default Root Certificates From Each Store

The most important step every IIS administrator must take to properly harden a box against the threat of improper use of certificate chains by other people (including Microsoft programmers) is to delete all third-party root CA certificates/Intermediate CA certificates. There is no good reason to automatically trust anything certified by any third-party CA, and the only reason for certificates to be installed in the Windows certificate stores is to allow automated trust determinations to be made by a variety of software programs and OS subsystems. You don't permanently remove all third-party certificates from every Windows box that you control, necessarily, because you will still want to conduct signature verification using a stand-alone forensic workstation prior to deploying code to production boxes. But any Windows box that is connected to a network should have all of its third-party certificates removed from default trust explicitly. The MMC Certificates Console snap-in makes this quick and easy.

Windows File Protection requires the Microsoft Root CA certificate to be present in addition to a couple other certificates in order to function. See Microsoft Knowledge Base Article Q293781 entitled "Trusted Root Certificates That Are Required By Windows 2000" but note that the actual list of root certificates truly required today is a subset of the list presented in Q293781. For maximum security tightening, remove all default root CA certificates and then add back those few that are required for features like WFP to work properly if you wish to use them.

Using The MMC Certificates Console Snap-In

Windows certificate stores live in the Windows Registry. Every subkey under the HKEY_USERS Registry hive including the .DEFAULT subkey that gets mapped into HKEY_CURRENT_USER when a security context has no interactive login profile available at run-time can contain separate certificate stores. Each certificate store is a subkey located under a SystemCertificates key like the following. The most important certificate stores are located under HKEY_LOCAL_MACHINE as shown.

HKEY_USERS\.DEFAULT\Software\Microsoft\SystemCertificates
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\SystemCertificates

System-wide default trusted certificates appear under the Registry hive HKEY_LOCAL_MACHINE, whereas user-specific certificates are found under the HKEY_USERS Registry hive. The Certificates Snap-In to MMC is designed to read and modify the certificate stores that exist in the Registry for each user account including .DEFAULT, for instances when it gets used by a security context, as well as for system-wide defaults. Figure 14-1 shows the Trusted Root Certification Authorities store under SystemCertificates in the HKEY_LOCAL_MACHINE hive.

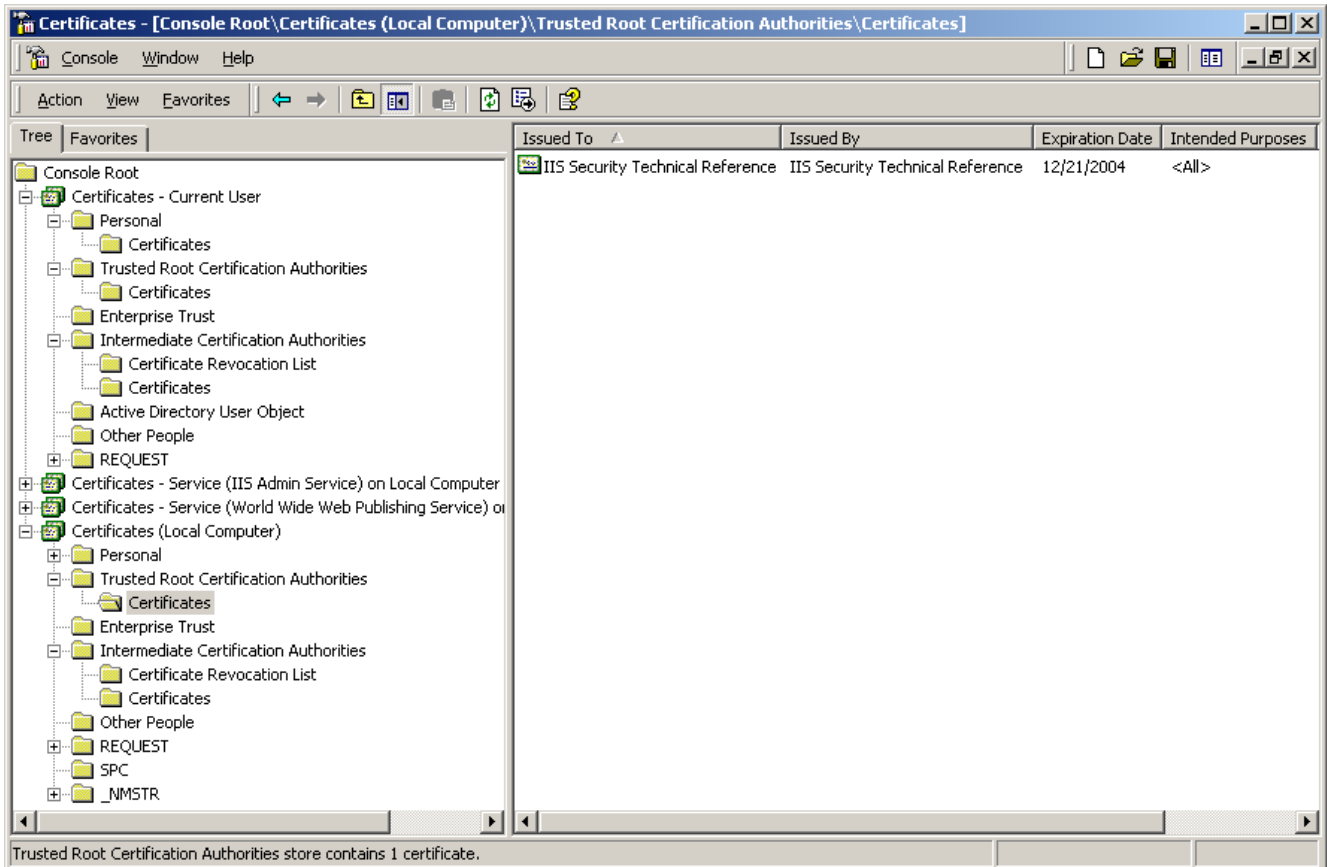


Figure 14-1: Use MMC Certificates Snap-In to View and Edit Certificate Stores on an IIS Box

To remove certificates from any certificate store using the Certificates MMC simply click on the certificates displayed for a store and press the delete key or choose Delete from the Action menu. It's a good idea to export the certificates that you plan to delete so that you have a record of the certificates that were previously installed in each certificate store. An Export option is located in the All Tasks menu.

Configuring Trusted Certificates for IISAdmin and W3SVC Services

Once you have removed the default certificates for each user and for the system under HKEY_LOCAL_MACHINE's SystemCertificates Registry key, you should also carefully examine the SystemCertificates that optionally exist for each local service. The optional certificate stores for services may contain trusted certificates other than those you might have configured explicitly in the system-wide certificate stores. Each service has its own certificate store, and the Certificates Snap-in to MMC is able to view and edit the certificate stores applicable to any local service. When you add the Certificates Snap-in you can select a service whose certificate stores you wish MMC to access. Figure 14-2 shows the Add Certificates Snap-in dialog for selecting a service for which to manage certificate stores.

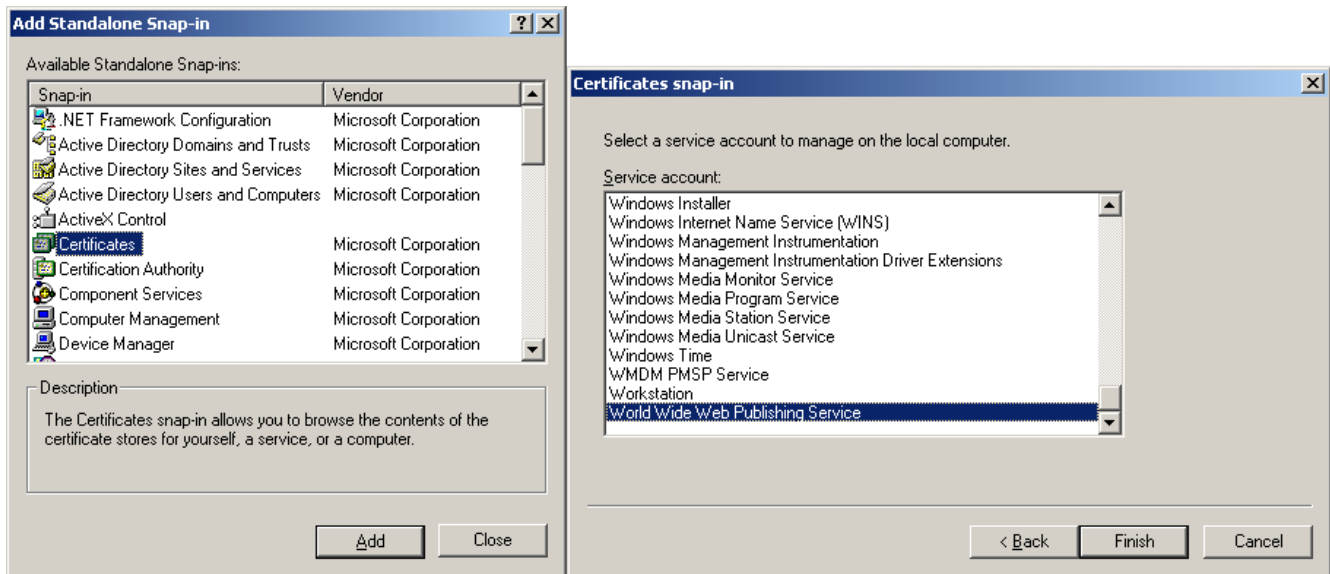


Figure 14-2: Certificates Snap-In Enables Management of Services' Certificate Stores

With a service account added as a distinct Certificates Snap-in instance in MMC you can now view and edit the certificate stores for a particular service. Figure 14-3 shows the W3SVC certificate stores, with the root certificates node selected. Each service's certificate stores exist as subkeys in the HKEY_LOCAL_MACHINE Registry hive at the following key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Cryptography\Services

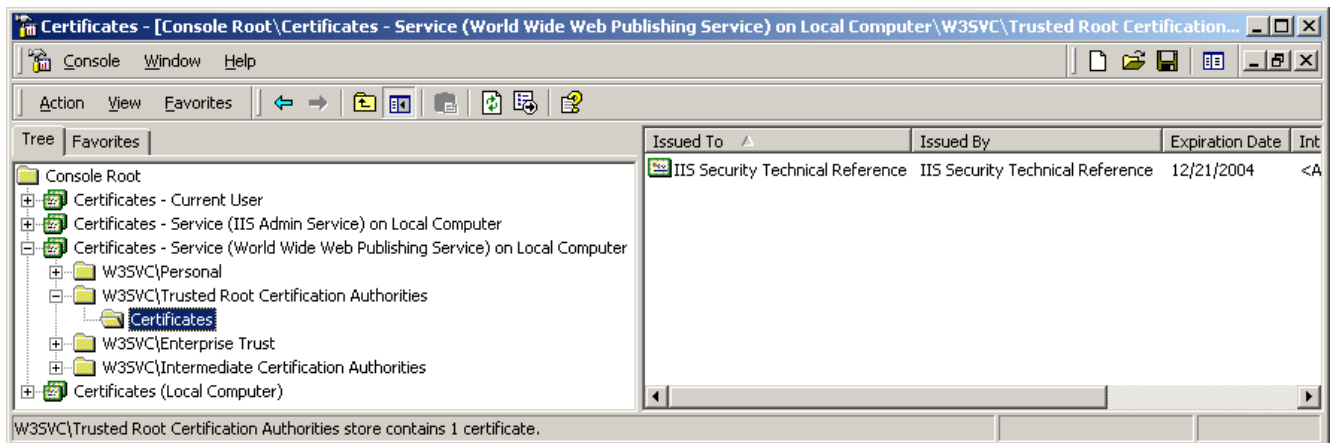


Figure 14-3: Trusted Root Certification Authorities for W3SVC

It's very important to prune the list of trusted certificates in each certificate store for each service when the service has its own Registry subkey under Cryptography\Services. By default each service relies on the system-wide default certificate stores, and adding a service Certificates Snap-In to MMC causes the creation of a corresponding Registry subkey under Cryptography\Services. The optimal configuration of certificate stores leaves the system default stores empty or populated with a single trusted root CA certificate issued by your own root CA while individual user account and service certificate stores contain explicit certificates. Each certificate configured in the system stores is incorporated implicitly into the list of explicit certificates present in the physical

certificate store of a user or service to form the logical certificate stores that are in effect for each user account or service.

Issuing Certificates with Windows Certificate Services

Some certificates are issued automatically by system services like Encrypting File System (EFS) or application programs like NetMeeting. In addition, certificate signing requests (CSRs) sent from a Windows box using CSR helper tools end up in the REQUEST certificate store of the requesting security context or service. In addition, certificates that are issued to a particular user are placed by default in the Personal store by most programs that install issued certificates under Windows. Figure 14-4 shows a typical Personal certificate store for Administrator on a box that uses EFS, client authentication, and functions as an issuing or root CA where the private key is kept online, in the encrypted protected storage of the administrator account, and under the administrator's exclusive control.

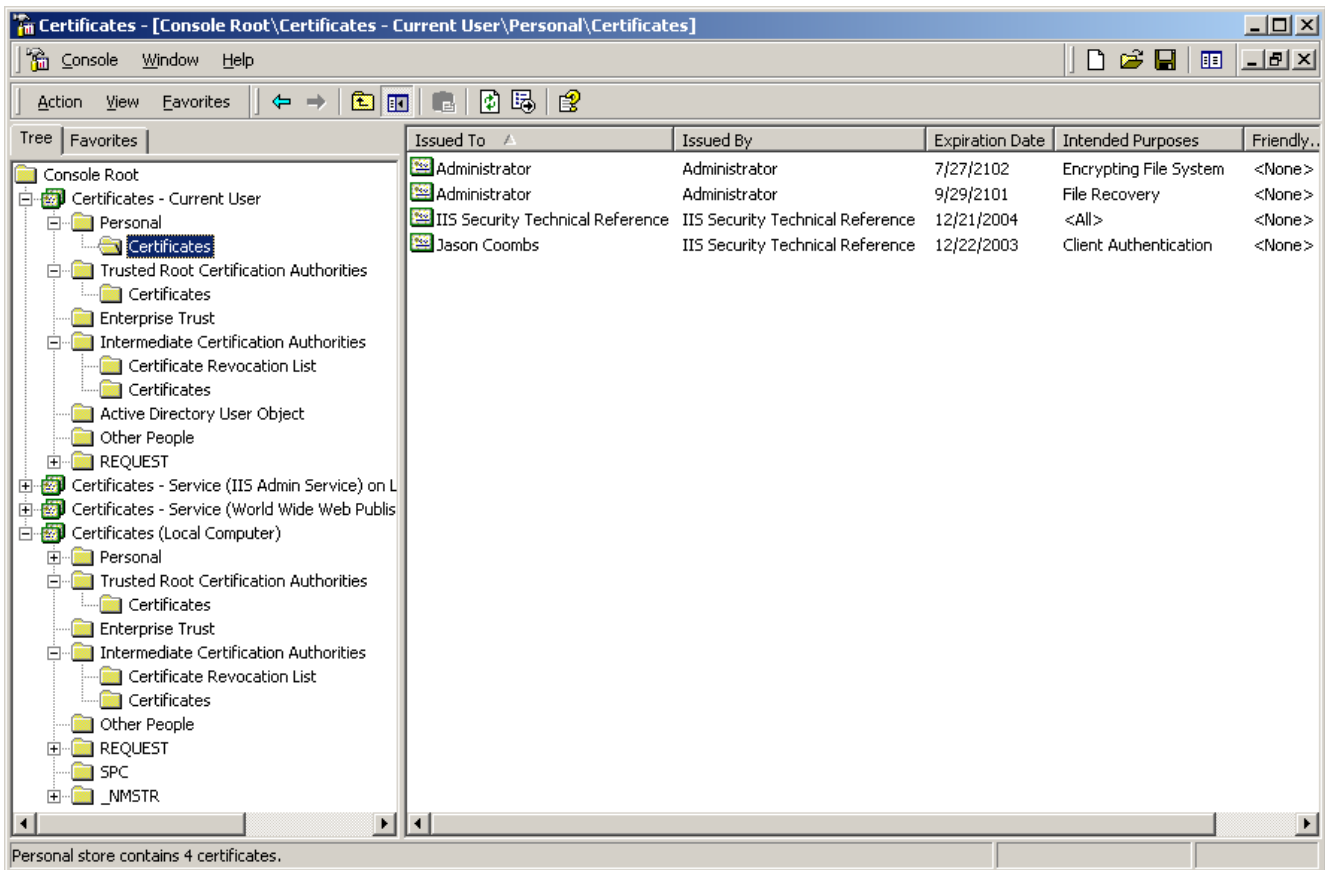


Figure 14-4: Personal Certificates Can Be Issued to Individual Users Explicitly

Encrypting File System automatically issues self-signed certificates with a NULL certificate chain for the purpose of encryption and key recovery by EFS users. Client certificates used for authenticating identity to servers or protected resources that require client certificate authentication are also found in each user's Personal certificate store. When Certificate Services is first configured as an optional Windows component, you select Stand-alone root CA as the type of Certification Authority to create, or Enterprise root CA if Active Directory is being used. Figure 14-5 shows the Windows Components Wizard

that walks you through setting up your Certification Authority. There is no need to install Certificate Services on a box that does not function as a CA; the full range of certificate store configuration options are available on every Windows box regardless, and Certificate Services plays no role in validating certificates at run-time or utilizing key pairs for encryption or generating or verifying digital signatures. Whatever you type in the CA name field becomes the Common Name, or CN, which appears in the Issuer field of all certificates issued by the CA.

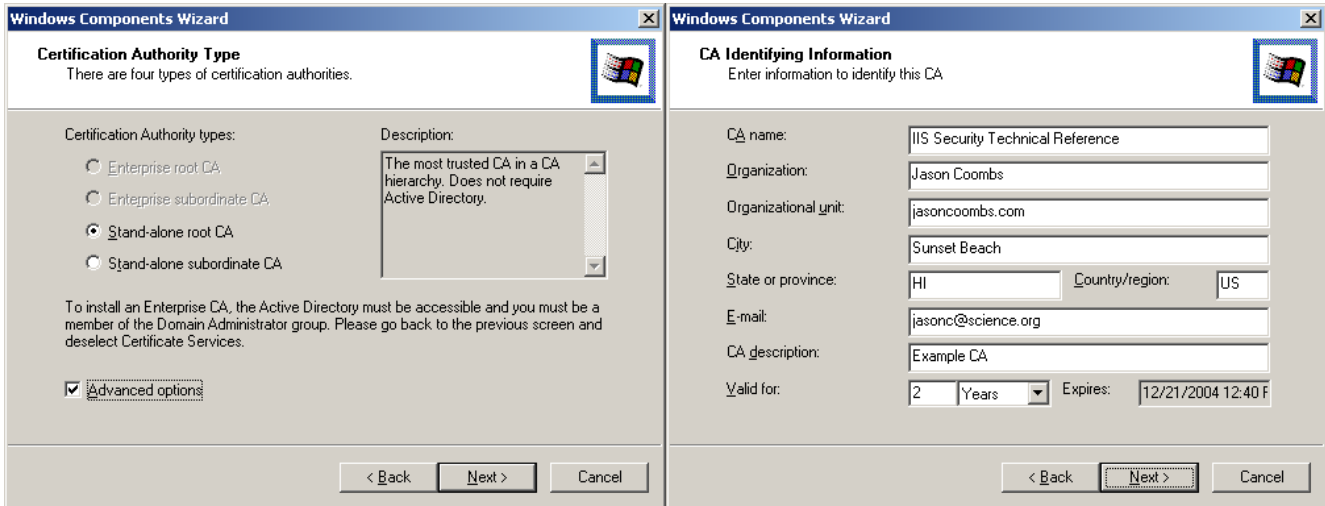


Figure 14-5: Configure Windows Certificate Services for a Private Stand-Alone Root CA

Your own private stand-alone root CA is the cornerstone of PKI for your IIS box or server farm. All automatic trust determinations made by each IIS box that you control must be made either based on a signature that appears to have been applied by your root CA (or a subordinate issuing CA) that is under your exclusive control, or based on validation of a digital signature applied by a trusted third-party based on a pre-defined public key that is known to correspond to that trusted third-party. The only certificate chains that should ever be used as the basis of automated trust are those that are issued and rooted at your own private CAs, and even this should be done only in special cases where the risk of certificate chain trust breach is limited.

Producing Certificate Signing Requests with Web Enrollment Support

A Certificate Signing Request (CSR) typically conforms to the PKCS #10 standard whose version 1.7 syntax is documented in RFC 2986. To produce a CSR the requesting entity applies a digital signature to their own public key as well as other information necessary for a CA to validate the CSR and construct a certificate. Because CSRs must be digitally signed, only the owner of a private key/public key pair can generate a CSR. When Active Directory is used in your network, CSRs can be created and sent automatically to any CA that is registered in the Windows Domain. Right-clicking on the Personal certificate store then selecting Request New Certificate from the All Tasks menu will bring up the Certificate Request Wizard. When a stand-alone rather than an enterprise CA is deployed instead, the CA is not available through Active Directory and certificate requests can be sent to the CA by hand using the Certificate Services Web Enrollment Support pages in a browser or by custom calls to Certificate Enrollment Control (CEnroll

implemented by xenroll.dll) and Microsoft Certificate Services Client (CCertRequest implemented by certcli.dll).

Web Enrollment Support is implemented as a collection of Active Server Pages scripts that instantiate these two components. The default virtual directory URL created for Certificate Services' Web Enrollment Support is part of the default Web site instance and is located at the following URL, where ServerName is your IIS box:

<http://ServerName/CertSrv/default.asp>

Sending Certificate Signing Requests with The CCertRequest Client

To construct PKCS #10-formatted CSRs explicitly from within program code you can use the Certificate Enrollment Control (ProgID of "CEnroll.CEnroll.1") supplied as a standard part of the Windows SDK. Certificate Services' Web Enrollment Support makes use of CEnroll therefore it is installed automatically whether or not you've installed the Platform SDK. In addition to creating CSRs with CEnroll, a Certificate Services Client (ProgID of "CertificateAuthority.Request" and known as CCertRequest in the SDK documentation) gives programmatic interface capability to Certificate Services for delivering CSRs and receiving certificates. These components are Automation compliant and can be instantiated from within VBScript hosted by Windows Script Host as well as within Active Server Pages. The following code sample shows how a simple command-line utility can be built using scripting to enable programmatic CSR and key pair creation plus request submission to a stand-alone Certification Authority provided by Microsoft Certificate Services.

(RFC 2986 PKCS #10 specification's URL is: <http://www.ietf.org/rfc/rfc2986.txt>)

```
dim cenroll
Set cenroll = CreateObject("CEnroll.CEnroll.1")
dim sCN, sDN, sOU, sO, sL, sS, sC, keyLen, sCA
if WScript.Arguments.Count = 8 then
  sCN = WScript.Arguments(0)
  sOU = WScript.Arguments(1)
  sO = WScript.Arguments(2)
  sL = WScript.Arguments(3)
  sS = WScript.Arguments(4)
  sC = WScript.Arguments(5)
  sDN = "CN=" & sCN & ",OU=" & sOU & ",O=" & sO
  sDN = sDN & ",L=" & sL & ",S=" & sS & ",C=" & sC
  keyLen = WScript.Arguments(6)
  sCA = WScript.Arguments(7)
  cenroll.GenKeyFlags = keyLen * 65536
  cenroll.PVKFileName = sCN & ".pvk"
  dim sPKCS10
  sPKCS10 = cenroll.createPKCS10(sDN,"1.3.6.1.5.5.7.3.1")
  dim careq
  Set careq = CreateObject("CertificateAuthority.Request")
  const CR_IN_BASE64 = &H1
```

```

const CR_IN_PKCS10 = &H100
dim REQ_FLAGS
REQ_FLAGS = CR_IN_BASE64 OR CR_IN_PKCS10
dim reqstatus
reqstatus = careq.Submit(REQ_FLAGS, sPKCS10, "", sCA)
else
WScript.Echo "Usage: certenroll [CN] [OU] [O] [L] [S] [C] [KEYLEN] [CA]"
end if

```

After creating a PKCS #10-formatted CSR, the code as shown creates a new instance of CertificateAuthority.Request and calls its Submit method to send the CSR to the specified CA. The final parameter on the command-line specifies the UNC path to a stand-alone or Active Directory-enabled CA in the form of "ServerName\CACN" where "CACN" matches the CN of the CA's root certificate. The CEnroll object's PVKFileName property specifies the relative path of the file in which the resulting private key will be written. If this property is not specified, then the private key is stored in the protected storage of the currently-active Windows user security context. The GenKeyFlags property of the CEnroll object is used to select the bit length of the key pair generated by CEnroll in a call to the CryptGenKey API function. The upper 16-bits of the 32-bit dwFlags parameter passed to CryptGenKey determines the key length produced by the API function, and multiplying the value passed to the script in [KEYLEN] as the seventh command-line parameter by 65536 (in binary 65536 is 1000000000000000) shifts the value for [KEYLEN] from the lower (right-most) 16-bits to the upper (left-most) 16-bits of the GenKeyFlags property.

The Distinguished Name (DN) of the certificate request created by CEnroll is specified by the first six command-line parameters. [CN] for Common Name, [OU] for Organizational Unit, [O] for Organization, [L] for City/Locality, [S] for State, and [C] for the two-letter Country code. A typical Distinguished Name looks like this:

```
CN=jasoncoombs.com,OU=Home,O=Me,L=Sunset Beach,S=HI,C=US
```

The CN of a certificate is especially important when a certificate is used for SSL encryption because the CN of an SSL certificate is compared against the FQDN of the server that the client contacts for the purpose of server identity authentication. A match indicates that the certificate was truly issued to the server being contacted and not issued to a different server. This automated server authentication process doesn't mean much else, and the rest of the certificate fields and its public key value must be examined too in order to determine whether or not a server authenticated automatically based on a CN/FQDN match really looks like the server it's supposed to look like. The public key is the most crucial identifying field of an SSL certificate and encountering a different public key than the one known to be associated with a particular SSL secured server should be the determining factor in whether or not automatic trust is given to a particular server based only on its certificate contents. Unfortunately, that's not the way most SSL client software works; any attacker who can obtain an SSL certificate with a CN that matches somebody else's FQDN can mount an impersonation or MITM attack with very little chance of detection. The change in public key from the authentic one known to be associated with the FQDN is proof enough of an attack in progress, but nobody bothers to look at this detail in the real world. Not even the people who operate the SSL-secured server, usually.

A certificate request is given a list of key usage Object Identifiers (OIDs) that determine for what purposes a certificate will be authorized for use. In order to have any meaning, the list of OIDs must be interpreted at run-time by software that enforces key usage restrictions. To minimize the chance of errors or software bugs in the way that CSR tools, CAs, and other PKI-compliant software interpret OIDs they are standardized using Abstract Syntax Notation 1 (ASN.1), an International Telecommunications Union (ITU) standard. Appendix B. entitled "1993 ASN.1 Structures and OIDs" in RFC 2459 which establishes the Internet X.509 Public Key Infrastructure Certificate and CRL Profile reserved 1.3.6.1.5.5.7 namespace prefix for PKI Internet security mechanisms. The ASN.1 OID of SSL Server Authentication is 1.3.6.1.5.5.7.3.1 and each key usage and certificate type is assigned a separate OID.

"Internet X.509 Public Key Infrastructure Certificate and CRL Profile" is defined in RFC 2459 and can be found online at <http://www.ietf.org/rfc/rfc2459.txt>

More information about the ASN.1 standard can be found at <http://www.asn1.org>

Algorithms, networks, attribute types, and an endless array of other structured data elements involved in electronic communications are all assigned unique ASN.1 identifiers to make it easier for computer systems to exchange complex formatted messages with specific purposes and meaning that should not be misinterpreted by the systems that process structured messages. When certificates and digital signatures are created and used by PKI systems, the ITU standard for PKI known as "module X.509 of the ASN.1 project" is typically followed. Module X.509 defines the format and content of PKI structured data along with OIDs for each algorithm and other possible value for PKI data fields used to construct things like certificates. When using CEnroll to create a CSR, the OIDs listing each acceptable use of the certified public key are supplied as a parameter to the createPKCS10 method. When more than one OID is listed, the OIDs are separated by commas.

Generating Your Own Code Signing Key Pairs and Certificates

Code Signing certificates contain enhanced key usage OID 1.3.6.1.5.5.7.3.3 in addition to other OIDs if desired by the requestor and approved by the CA when the certificate is issued. A code signing certificate issued by your own private root CA is one of the most important security devices available to protect your IIS box from malicious compromise. Applying your own signature to trusted code and configuring your IIS box to require your digital signature to validate that trust is one of the most reliable ways to control unwanted code and distinguish between code that you've reviewed forensically and arbitrary code installed without your full consent by a vendor's setup program. You can easily create a code signing certificate using either Web Enrollment or the CCertRequest client and CEnroll.

To create a key pair, CSR, and certificate for code signing through script like that shown in the previous code sample that uses CEnroll and CCertRequest, simply replace the SSL Server Authentication OID 1.3.6.1.5.5.7.3.1 with the Code Signing OID 1.3.6.1.5.5.7.3.3 instead. Alternatively, to produce a certificate with both SSL Server Authentication and Code Signing key usage rights, supply both OIDs like this:

```
createPKCS10(sDN,"1.3.6.1.5.5.7.3.1","1.3.6.1.5.5.7.3.3")
```

It's important to note that key usage and basic constraints OIDs are merely informational. They are often ignored purposefully or by mistake by software that makes use of certificates for encryption and trust. Since you can't know in advance that other people's software will always implement the same security policy rules and interpretations of the intended impact of certain OIDs, you can never assume that the presence or lack thereof of any constraint or key usage OID will really have the impact that you expect it to have everywhere that a certificate might be used.

Trust Only Your Own Digital Signatures and Root CAs

For optimal security with PKI you should only trust your own digital signatures issued by your own root CAs. The fewer third-party CAs that you trust on production IIS boxes to certify public keys belonging to other third-parties the better. Most importantly, any third-party who wants you to trust them to deploy code to your IIS boxes should ask permission explicitly by submitting a CSR to your CA so that you can issue your own certificate chain that certifies the third-party's public key. PKCS #10 doesn't call for CSRs to contain any information identifying the CA to which the signing request is submitted, therefore anyone who uses PKI to digitally sign code should publish a PKCS #10 CSR that you can submit to a CA of your choice to obtain a certificate signed by your root CA or a third-party CA that you choose to trust rather than being required to accept and deploy the certificate chain offered by the software publisher signed by the CA that they choose to trust. Design limitations of Windows PKI trust management make it unsafe to trust root CA certificates belonging to third parties because there is no way to selectively apply and control trust that extends from installed CA certificates. Any CA certificate active on your IIS box is trusted fully for all potential uses of certificate chains issued by that CA. And in many cases this full trust by default results in automated trust determinations made by Windows and its subsystems without allowing any human to approve or reject individual certificates.

Signing and Timestamping Trusted Windows Binaries or Custom Code

Windows' Platform SDK includes a utility called the Digital Signature Wizard (Signcode.exe) for applying digital signatures to PE files, Security Catalog (.CAT) files, Cabinet (.CAB) files, and Certificate Trust List (.STL) files. Digital signatures are applied directly to PE/COFF and other file types in Windows with the help of SIP providers for hashing and parsing of attached signature block data. Any file can be digitally signed using standard PKI data structures like those defined by X.509 if the signature block remains detached and is transported independent of the signed file, but when signature blocks are attached to files a new formatting problem emerges that is handled differently depending on operating system platform and file type. Flat files that don't conform to a structured file type for which a SIP provider is installed and registered with the Windows Crypto API can't be digitally signed by the Digital Signature Wizard because there is no way to attach the resulting signature block to the data without destroying the original file format making it unusable by applications that understand the original flat file data format. Figure 14-6 shows the signcode.exe user interface that walks you through signing files. In addition to the Wizard UI, signcode.exe supports command-line parameters for signing files without the Wizard.

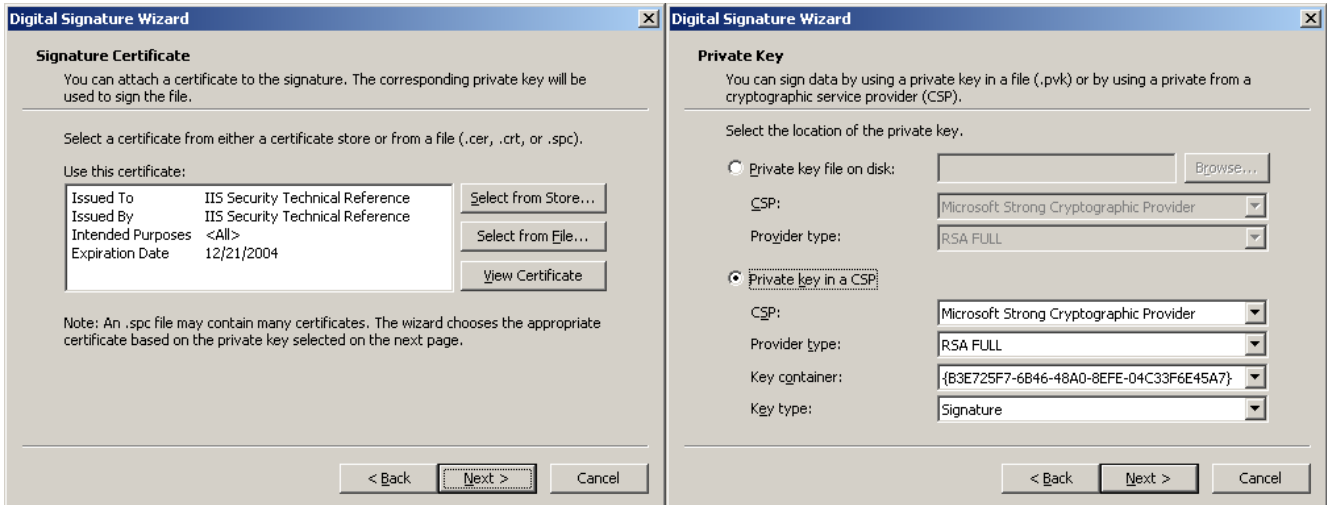


Figure 14-6: Select Public Key Certificate and Private Key Location in Digital Signature Wizard

Each digital signature can optionally contain copies of the certificates that make up the signature’s supporting certificate chain as well as a countersignature created by signing a timestamp. A third-party timestamping service can be used provided that the signature verification platform to which a digitally signed and timestamped file is deployed is likely to trust the timestamping service’s root CA certificate. Figure 14-7 shows the certificate chain and timestamp service URL configuration settings options available in the signcode.exe Wizard UI.

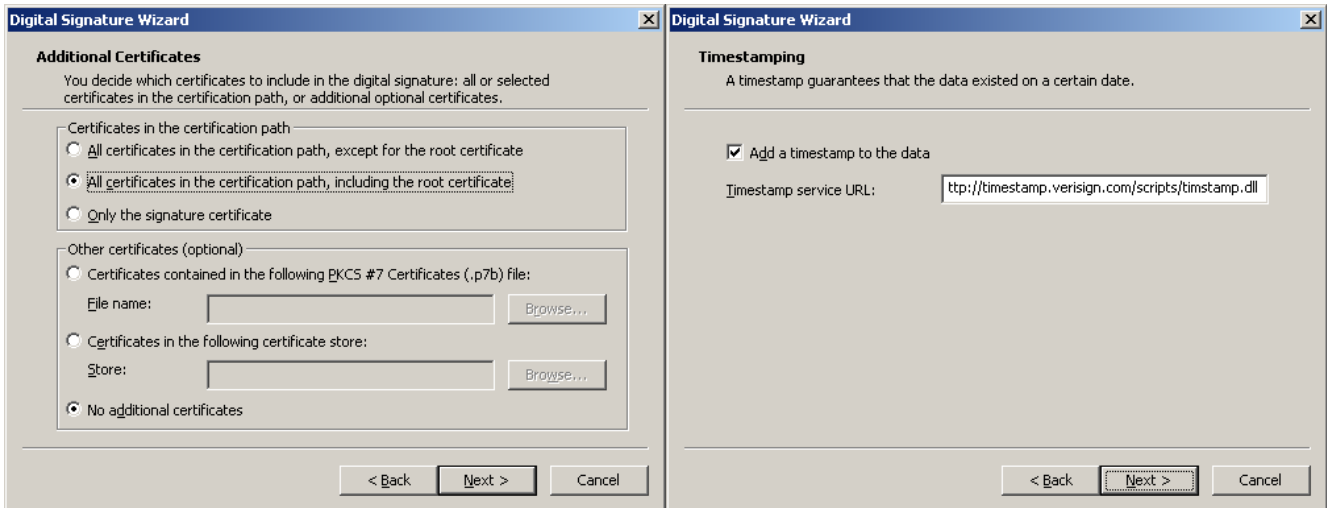


Figure 14-7: Specify Certificate Chain Preference and Optional Timestamp Service URL

When a digital signature is verified, its countersignatures are also verified, if any are applied in addition to the primary signature. If the signature and timestamp verifier doesn’t choose to trust the same timestamping root CA that you rely on to produce timestamp countersignatures, the digital signature will still appear to be valid but the timestamp signature won’t be verifiable. VeriSign operates a publicly-accessible timestamping service that you can use to apply a timestamping countersignature to signed files on-demand and free of charge. The URL is:

<http://timestamp.verisign.com/scripts/timestamp.dll>

Windows File Protection Arbitrary Certificate Chain Vulnerability

Windows File Protection (a.k.a. Windows Driver Signing) verifies digital signatures applied to operating system binaries, device drivers, and other OS files, as well as files published by third-parties that are certified by Windows Hardware Quality Labs (WHQL) (a.k.a. Microsoft Windows Hardware Compatibility). There is a vulnerability in WFP that causes any digitally-signed replacement file of malicious origin to take priority over any authentic WFP/WHQL-signed file. This vulnerability extends to every file protected by Windows File Protection including Security Catalog (.CAT) files. More than just of academic concern, this vulnerability enables the replacement of authentic Microsoft Windows binaries with arbitrary malicious code in such a way that the replacement code is verified automatically as trustworthy by WFP digital signature verification. Anyone can now obtain anonymous code signing and SSL certificates automatically and free of charge from the following CA:

GeoTrust, Inc.
<http://www.freessl.com>

The Root Certificate for GeoTrust's FreeSSL CA is:

CN = UTN-USERFirst-Network Applications

OU = <http://www.usertrust.com>

O = The USERTRUST Network

L = Salt Lake City

S = UT

C = US

Anyone who controls a DNS domain can obtain a certificate from FreeSSL that can be used for code signing and use this essentially anonymous certificate to digitally sign malicious code (e.g. using SIGNCODE.EXE) that WFP will automatically trust by virtue of the fact that the certificate's Root CA (usertrust.com) is one of the Root Certificates trusted by default in standard Windows/IE installations. It should be noted, however, that every Root CA that issues certificates that can be used for code signing enables any attacker in possession of such a certificate to apply digital signatures to malicious code files and deploy them without detection to any Windows box that relies on WFP for protection against Trojans. WFP trusts signed files automatically based on the presence of verifiable arbitrary certificate chains based on any root certificate that is apparently signed by any trusted Root CA.

Information about Driver Signing for Windows can be found at these URLs:

<http://www.microsoft.com/hwdev/driver/digitsign.asp>

<http://www.microsoft.com/hwdev/driver/drvsign.asp>

The only way to use Windows File Protection safely is to apply your own digital signatures to each of the files you would like WFP to protect or trust. As long as your IIS box is configured to trust only your own root CA certificate, other people will be unable to produce digital signatures that your IIS box can validate automatically unless your CA's

private key is also compromised. The security alert notice that details this vulnerability in WFP can be found at the following address:

WFP Arbitrary Certificate Chain Vulnerability

<http://www.forensics.org/secalert/>

Alternatively, you can disable WFP and build your own file protection mechanism based on digital signatures by scripting the verification of trust using CHKTRUST.EXE instead of WFP, since CHKTRUST.EXE relies on the WinTrust API instead of WFP. WinTrust will only trust software publisher certificates (SPCs) that are selected explicitly and configured for automatic trust with Registry keys:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\WinTrust\Trust
Providers\Software Publishing\Trust Database
HKEY_USERS\DEFAULT\Software\Microsoft\Windows\CurrentVersion\WinTrust\Trust
Providers\Software Publishing\Trust Database
```

There is no reason to automatically trust any Root CA when it comes to code signing.

Windows File Protection should never have been designed to trust digital signatures on software based on certificate chains, WFP should trust only specific certificates the way that WinTrust operates. It would be sensible to also deploy a better system for managing certificates and trust relationships with End Entities, Root CA's, and any site that needs SSL for the purpose of encryption but doesn't care about the authentication features. There is a large and legitimate demand for anonymous SSL certificates like those distributed by GeoTrust FreeSSL, however, bad code deployed in the wild today like Windows File Protection, and flawed security policies that rely on such bad code, make the availability of free, anonymous SSL certificates or code signing certificates an urgent and immediate information security threat.

Ideally, certificates would be managed by each node or end-user in a manner similar to the way that cookies are now managed in Internet Explorer. Each end-user or administrator of each node on the network should be able to easily define a security policy and the default setting should be to block and deny all. Each capability possible with respect to each certificate (i.e. SSL/code signing/e-mail signatures, etc.) should be a separate security policy setting that the end-user or an authorized administrator must explicitly allow on a per-certificate basis. The closest Windows comes today to achieving this level of configurable trust settings for certificates is the ability to define specific Certificate Trust Lists (CTLs) which are collections of root certificates grouped together in a single digitally-signed .STL file.

Windows File Protection Old Security Catalog Vulnerability

Old Security Catalogs (.CAT files) containing valid digital signatures are left in place under %WinDir%\System32\CatRoot when new files and their associated Security Catalogs are deployed. The Windows API function discussed in Chapter 13 for computing file hashes, CryptCATAdminCalcHashFromFileHandle, is used by WFP to compute a file's SIP hash code and then a related API function named CryptCATAdminEnumCatalogFromHash is

called to automatically locate the digitally-signed .CAT files, if any, that contain the file's SIP hash code. If a file's SIP hash code cannot be found inside any digitally-signed .CAT file and the file itself contains no digital signature then the file is considered to be "unsigned". Windows File Protection gives the same priority and preference to authentic hash codes of old binaries (and other protected files) as it does to authentic hash codes of newer, updated binaries. An attacker can therefore place old authentic files containing known security vulnerabilities in place of newer files from hotfixes and service packs and WFP will automatically trust and certify the authenticity of the older files.

To see this work, use a version of Windows known to be vulnerable to this WFP bug and copy and paste an old version of a protected file first to the dllcache directory under system32 and next to the full path where the file normally lives.

To enable multiple trust authorities to certify the same files independently without altering the hash code computed by WHQL for its Windows Hardware Compatibility signature, WFP relies on a proprietary Subject Interface Package (SIP) object file hashing mechanism that applies hash algorithms to a subset of the bits contained in any Portable Executable (PE) file rather than to the entire file through a full-file hashing mechanism. This SIP hashing mechanism for PE files also enables a software vendor to localize language- and locale-specific versions of each file without altering the object's SIP hash code when localizable strings are stored in a resource header rather than hard-coded inside object code. If any portion of the compiled object code changes, then the file's SIP hash code changes as well, which would invalidate the hashes contained in signed Security Catalogs. The "Certificates Table" data directory entry in an executable's IMAGE_DATA_DIRECTORY table located at the end of its PE header IMAGE_OPTIONAL_HEADER structure is excluded from the hashed bits by the SIP object file hash preprocessor module. Every PE file can thus have digital signatures attached at-will in a production system without invalidating the file's known good trusted SIP hash code as certified by a digitally-signed Security Catalog (.CAT) file. WFP uses SIP hashes to avoid the deployment hassle caused by the variability of full-file hashes when files are localized for international use or when the owner of a Windows box applies digital signatures directly to PE files previously certified as authentic by a software vendor.

To simplify the process of code signing, so that every file need not be signed individually and updated signatures can be deployed at run-time (e.g. when certificates expire or private keys become compromised) without replacing files that might be in use (and thus locked for writing) Windows File Protection uses Security Catalogs (.CAT files) that are digitally-signed. Each .CAT file contains a list of authentic SIP hashes of trusted files that Windows File Protection (via SFC.EXE and SIGVERIF.EXE as well as automatic protection feature) considers to be valid SIP hashes. Every file that, when hashed with the help of the default PE SIP provider, results in a SIP hash code that is contained in any .CAT file is considered trustworthy by Windows File Protection, even if updates to the file have been deployed and the newest version of the authentic code in fact contains a different SIP hash from the one that Windows File Protection encounters.

Delete the Security Catalogs (.CAT files) provided by your vendors. Produce your own instead, and sign them with a code signing certificate that you issued to yourself from your own Trusted Root Certification Authority certificate store. There is no reason to let

mutually-exclusive Security Catalog files exist in production systems. Doing so results in a vulnerability when somebody is able to tamper with a Windows box on purpose. Shipping a hotfix or service pack with a new Security Catalog file without a mechanism to remove the out-of-date .CAT file(s) when the new ones are installed defeats a core purpose of Windows File Protection entirely: simplifying the process of distributing authentic hashes. Even Windows File Protection is unable to determine which of the SIP hashes is the most-current "authentic hash" of a given file. A redesign of this whole process will most likely occur in the future, with enhancements to the Security Catalog file format. In the mean time, you can make this vulnerability in Windows File Protection irrelevant to the security and integrity of your IIS box by applying your own digital signatures directly to the protected files that you choose to trust.

Creating and Signing Your Own Security Catalog Files

Before you can create and sign your own Security Catalogs, you must create a Catalog Definition File (.CDF) that lists the files to be hashed and the display name (tag) that will appear alongside each file hash inside the .CAT file. After you've created a .CDF, the MAKECAT.EXE utility reads a .CDF input file and attempts to build a .CAT file based on the [CatalogHeader] instructions and [CatalogFiles] file list it finds inside. A sample .CDF file is shown below. The <HASH> prefix before each file listing indicates that the tag (display name) assigned to each file in the catalog will be the hex-encoded hash value rather than the filename or other identifier present to the left of the "=" on each [CatalogFiles] line.

```
[CatalogHeader]
Name=iis.cat
ResultDir=.
PublicVersion=0x00000001
[CatalogFiles]
<HASH>asp=.asp.dll
<HASH>FTPSVC2=.FTPSVC2.DLL
<HASH>IISADMIN=.IISADMIN.DLL
<HASH>IISLOG=.IISLOG.DLL
<HASH>inetinfo=.inetinfo.exe
<HASH>W3SVC=.W3SVC.DLL
<HASH>wam=.wam.dll
```

Once your Security Catalog file is created, you should apply a digital signature to it using SIGNCODE.EXE and your code signing key pair for which you've issued a certificate rooted at your private CA. The Crypto API provides a couple of functions to help with installation of .CAT files into the Security Catalog subsystems' storage. By default, Security Catalogs are placed in the directory %windir%\System32\CatRoot in a subdirectory named with the GUID assigned to a particular subsystem, or the default subsystem. The following code shows a simple command-line utility written in C++ that performs default subsystem installation. The code shown here calls four previously undocumented WinTrust API functions that are used for Security Catalog file management, CryptCATAdminAcquireContext, CryptCATAdminAddCatalog, CryptCATAdminReleaseCatalogContext, and CryptCATAdminReleaseContext. Microsoft released documentation for these APIs in August of 2002 as part of the U.S. Justice

Department antitrust settlement agreement. More information about these APIs can now be found in the Microsoft SDK.

```
#define WIN32_LEAN_AND_MEAN
#include <stdio.h>
#include <tchar.h>
#include <windows.h>
typedef HANDLE HCATADMIN;
typedef HANDLE HCATINFO;
typedef BOOL (WINAPI * PFN_AACONTEXT)(OUT HCATADMIN
 *phCatAdmin, IN const GUID *pgSubsystem,
 IN DWORD dwFlags);
typedef HCATINFO (WINAPI * PFN_AACATALOG)(IN HCATADMIN
 hCatAdmin, IN WCHAR *pwszCatalogFile,
 IN OPTIONAL WCHAR *pwszSelectBaseName,
 IN DWORD dwFlags);
typedef BOOL (WINAPI * PFN_ARC)(IN HCATADMIN hCatAdmin,
 IN DWORD dwFlags);
typedef BOOL (WINAPI * PFN_ARCC)(IN HCATADMIN hCatAdmin,
 IN HCATINFO hCatInfo, IN DWORD dwFlags);
int _tmain(int argc, _TCHAR* argv[]) {
    char * filename = NULL;
    HCATADMIN hcatA;
    HCATINFO hcatI;
    WCHAR wfilename[MAX_PATH];
    size_t len = 0;
    int iWChars = 0, iErr = 0;
    HMODULE hm = LoadLibrary("mscat32.dll");
    PFN_AACONTEXT f1 = (PFN_AACONTEXT)GetProcAddress(
        hm,"CryptCATAdminAcquireContext");
    PFN_AACATALOG f2 = (PFN_AACATALOG)GetProcAddress(
        hm,"CryptCATAdminAddCatalog");
    PFN_ARCC f3 = (PFN_ARCC)GetProcAddress(
        hm,"CryptCATAdminReleaseCatalogContext");
    PFN_ARC f4 = (PFN_ARC)GetProcAddress(
        hm,"CryptCATAdminReleaseContext");
    if(argc == 2 && f1(&hcatA,NULL,NULL)) {
        filename = argv[1];
        len = strlen(filename);
        if(len <= MAX_PATH) {
            iWChars = MultiByteToWideChar(
                CP_ACP,NULL,filename,len,wfilename,MAX_PATH);
            wfilename[iWChars] = (WCHAR)NULL;
            hcatI = f2(hcatA,wfilename,NULL,NULL);
            if(hcatI != NULL) {
                printf("Security Catalog Added to Default Subsystem\n");
                f3(hcatA,hcatI,NULL); }
            else { iErr = GetLastError();
                if(iErr == ERROR_BAD_FORMAT) {
```

```
printf("ERROR_BAD_FORMAT: not a catalog file\n"); }  
else { printf("ERROR: %d",iErr); }}  
f4(hcatA,NULL); }}  
FreeLibrary(hm);  
return 0; }
```

For your digitally-signed .CAT file to be most useful, you will want it to be compatible with Windows File Protection. Without this compatibility your .CAT file is useful only to document the authentic SIP hashes of the trusted code that you've deployed to your IIS box. Although any text editor can be used to create a Catalog Definition File (.CDF) and the MAKECAT.EXE and SIGNCODE.EXE utilities are self-explanatory and free as a part of the Windows SDK, to get your Security Catalogs to be trusted properly by Windows File Protection you will need to contact Microsoft's Windows Hardware Quality Labs (WHQL) about obtaining a Microsoft digital signature for your custom security catalogs. The WHQL Web site URL is:

<https://winqual.microsoft.com/>

Side-by-side Assemblies are a new feature of .NET that provides additional functionality for security catalogs and code signing. Three utilities are provided for use creating and deploying .NET assemblies that are potentially useful for other security catalog and code signing operations as well. To sign a file and create a security catalog in one step, a utility called MT.EXE is provided that reads an input assembly manifest, builds a .CDF file from it, and then invokes MAKECAT.EXE to write the .CAT file. Finally, PKTEXTRACT.EXE is a utility that extracts public keys from certificate files, something that is useful for a variety of reasons including auditing and verifying public keys contained within signed certificates or encrypting data that only the public key owner can decipher.

Designing a Multi-Level Certification Hierarchy

Most guides to PKI recommend the creation of a hierarchy of trust that has multiple levels starting with one or more off-line (powered-down and secured, ideally inside a vault or under a mattress) root CAs. The off-line root CA is brought on-line (powered-up, but never connected to a network) once every twenty years or so when new certificates have to be issued for intermediate CAs. Intermediate CAs are booted more frequently, but are also kept off-line until they are needed to create or renew certificates for issuing CAs.

While issuing CAs might be left on-line all the time, and may even issue certificates automatically to anyone or anything who wants one. The main reason for multiple levels in this sort of trust hierarchy is to create fire breaks that contain the damage done when a particular unit of trust is compromised. There are other reasons for a hierarchy, such as the ability to distribute administrative authority for issuing certificates to many people while configuring computer systems that are all under the control of one entity to trust any certificate issued by any one of the entity's authorized representatives. Deploying many issuing CAs also works as a built-in load-balancer, since any one of the issuing CAs has the ability to service a request for a certificate. For most applications in the real world this is all nonsense.

The hierarchy of trust that you allow your IIS box (and its client nodes, if they are also under your security control) to rely on for authenticating public keys offered by entities that request trust is one of the trickiest things to manage safely in PKI. Recall that certificate chains are optional in the first place, and that it is inappropriate to allow a computer to automatically verify arbitrary certificate chains, which are properly validated only by human beings with the help of trustworthy software. What point is there, then, in having any CA at all, much less a chain of intermediate and issuing CAs? On a practical level, you have to have at least one CA because many PKI software tools used for producing certificates won't create a certificate with a NULL certificate chain, and such a certificate often won't be considered a valid certificate automatically by other PKI software. Does a valid certificate chain make a certificate more trustworthy? Not really, it's just another subjective reference point that can be used to ascertain a probability of safety, like whether or not a person who walks into your store as a prospective customer is carrying a firearm, and whether or not the person points the firearm at you or whether it remains in a holster. The apparent validity of a certificate chain may help you to predict future dangerous events more accurately, but it doesn't convey objective forensic proof of anything.

If your life and your personal freedom are at risk because a person in possession of a certificate that had your name on it did something criminal and you got blamed, would you be satisfied that the certificate had a valid certificate chain and therefore accept that you must be guilty? Of course not, you would demand a detailed examination of the public key itself to find out if it matches yours. Then you would search for evidence that your private key had been stolen. Then you would examine every detail of the software system that was used to find out if buffer overflow vulnerabilities or other flaws might have enabled somebody to trick the system into believing they had possession of your private key when in fact they did not, or compel the system to skip this validation step entirely, rely on a different private key, or just go and do something it isn't supposed to do that benefits the attacker and points the finger of blame at you. When information security really matters, you will accept nothing less than forensic proof. Why accept anything less at any time?

With this caveat in mind, it's important to recognize that we all take risk every day of our lives that we consider to be reasonable. People have different concepts of reasonable risk. Information security vulnerabilities are really just risk factors that impact people who own and operate certain products built by certain vendors or that impact algorithms and architectures selected by those who build products. The risk a person is willing to take and the very understanding of risk are fluid and flexible things in the minds of every person.

PKI can convey this same fluidity, flexibility, and ability to adapt to changing risk conditions and changing risk factors to information systems. Without a multi-level PKI hierarchy, risk assessments and controls are much more black and white, and it isn't nearly as easy to invalidate just a portion of a trust assumption and replace it with something else, we're stuck reissuing every user account a new password or taking other all-or-nothing actions. Real life doesn't work this way; you wouldn't divorce your spouse or end relationship with a friend because they can't prove beyond any doubt where they are or what they are doing every moment they are away from you. At the same time, however, you don't authenticate your spouse based on an arbitrary certificate chain and then remark "My,

what big eyes you have; my, what a big nose you have; my, what big teeth you have;” then wonder why you never noticed these things before and go to sleep. There is no harm in managing your private CA trust web with a multi-layer hierarchy and issuing certificates for trusted entities from a matrix of trust rather than a single root issuer, provided that you put in the prerequisite effort to manage this more complex trust network and properly train each person who manages a portion of it.

Client Certificate Authentication

Client certificates can provide far better security than password credentials alone can offer. A client certificate binds a known public key to a particular user who controls the corresponding private key. To make this type of authentication work properly in IIS you must know in advance either the actual certificate that each user will present so that you can configure a one-to-one mapping between Windows user accounts and client certificates, or you must know something about the Subject or Issuer portion of a group of certificates. For example, you might know in advance that every client certificate used by employees of your company contains the same “O=” specifying the name of your organization. As long as you control the trust chains that are potentially valid from the perspective of your IIS box and issue client certificates yourself from your own private CA, you can configure many-to-one mappings between many client certificates that share common features and a single Windows user account that can serve as the native security context for request processing on behalf of a community of users. These mappings start with issuing client certificates.

Issuing Client Certificates with Web Enrollment

The Active Server Pages script that implements Web Enrollment is provided in script source and raw HTML format that you can customize to your liking. The CEnroll and CCertRequest objects (“CEnroll.CEnroll.1” and “CertificateAuthority.Request”) implement COM interfaces named ICEnroll and ICertRequest, respectively. Neither these interfaces nor the ASP script source have been security-hardened, necessarily, as they are not meant to be accessed directly by untrustworthy end users. For advanced users and administrators you can simply give access to the built-in Web Enrollment pages supported by Certificate Services. Web Enrollment is a virtual directory in IIS that points at the %windir%\System32\CertSrv physical directory. To setup the certsrv virtual directory on the default Web site instance in IIS, issue the following certutil command using the command prompt:

```
certutil.exe -vroot
```

To remove the certsrv virtual directory, issue the following command:

```
certutil.exe -vroot delete
```

The preferred method of exposing certificate request and processing services to users of IIS applications is to code your own hardened Web interface. Secure automated certificate request processing with custom Web pages and server-side scripts can be accomplished easily when the issuing CA is on-line at all times accessible to IIS or configured on the same box as IIS. To issue client authentication certificates automatically to Web site

users you must configure Certificate Services to receive, process, and issue certificates automatically without human intervention. Figure 14-8 shows the two different options for certificate issuance policy that are configurable in the Policy Module Configuration Properties window. Access this configuration setting by choosing Properties from the Action menu or right-clicking on the CA within the Certification Authority tool for managing Certificate Services.

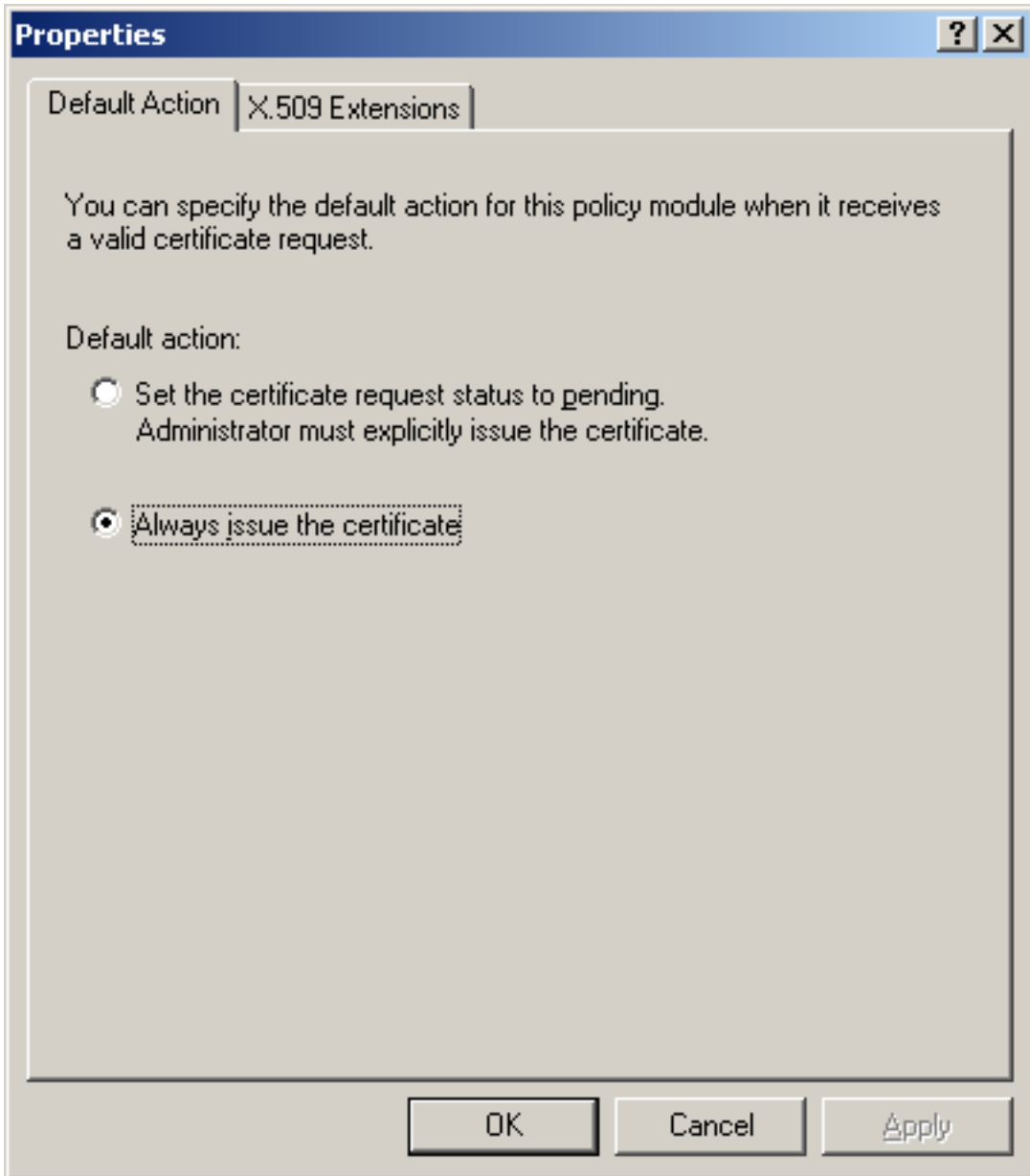


Figure 14-8: Explicitly Issue or Always Issue Option in Policy Module Configuration Properties

When CEnroll is used to generate key pairs on the server rather than requiring clients to generate their own key pair leaves end users' private keys vulnerable to theft. Any Web application administrator can also keep a copy of the private keys issued to end users, making the key pairs unusable for non-repudiation and reliable digital signatures.

However, when members of your user community don't have the ability to generate their own key pair, using CEnroll to create a key pair might be an acceptable risk in your deployment scenario. For optimal security, end users must create their own key pair and PKCS #10 CSR then deliver the CSR to the Web interface for an issuing CA. Delivering a key pair over the network to end users doesn't offer optimal security but it nonetheless provides far greater client identity authentication assurance than any password-based session-oriented mechanism such as ASP.NET Forms Authentication- or HTTP Basic Authentication-based client authentication mechanisms. A certificate issued to certify a public key from a key pair generated by your server on behalf of end users must be even more careful about assigning broad key usage OIDs because allowing arbitrary digital signatures to be produced and encryption to be performed with the help of a valid certificate for these purposes issued by your CA is inappropriate considering the origin of the key pair. In particular, none of the Microsoft OIDs that start with 1.3.6.1.4.1.311 should normally be included in a certificate issued under such circumstances.

Knowledge Base Article Q287547 entitled "Object IDs Associated with Microsoft Cryptography" lists 1.3.6.1.4.1.311 ASN.1 OIDs reserved for Microsoft Crypto

The only OID that is appropriate for most client authentication certificates is the OID assigned by RFC 2459 for SSL Web Client authentication: 1.3.6.1.5.5.7.3.1. Additionally, your custom Web interface to facilitate key pair generation and download by end users along with download of client certificates issued by your CA should allow the end user to control few, if any, of the values of Distinguished Name (DN) fields. If your Web application allows user accounts to be created of the end user's arbitrary selection, do not use the user account name as the certificate Common Name (CN). Instead, generate a unique CN that the user cannot control and map the certificate issued for the end user to that user's account on the server. Whether or not your server generates the key pair for end users to use in conjunction with client certificates issued by your CA, limiting the key usage OID to 1.3.6.1.5.5.7.3.1 is important in order to prevent the certificates thus issued from being trusted for any purpose other than client authentication by other PKI-aware software systems.

Mapping Client Certificates to User Accounts

The configuration settings that control the way in which client certificates submitted by Web site end-users are mapped to Windows user accounts are accessible only after you enable SSL for the Web site. Click on the Edit button in the Directory Security tab of the Web site properties window to open the Secure Communications settings shown in Figure 14-9. Select Require secure channel if you wish to Require client certificates, and then select Enable client certificate mapping. Click the Edit button to configure mappings in one of two ways: many-to-one or one-to-one.

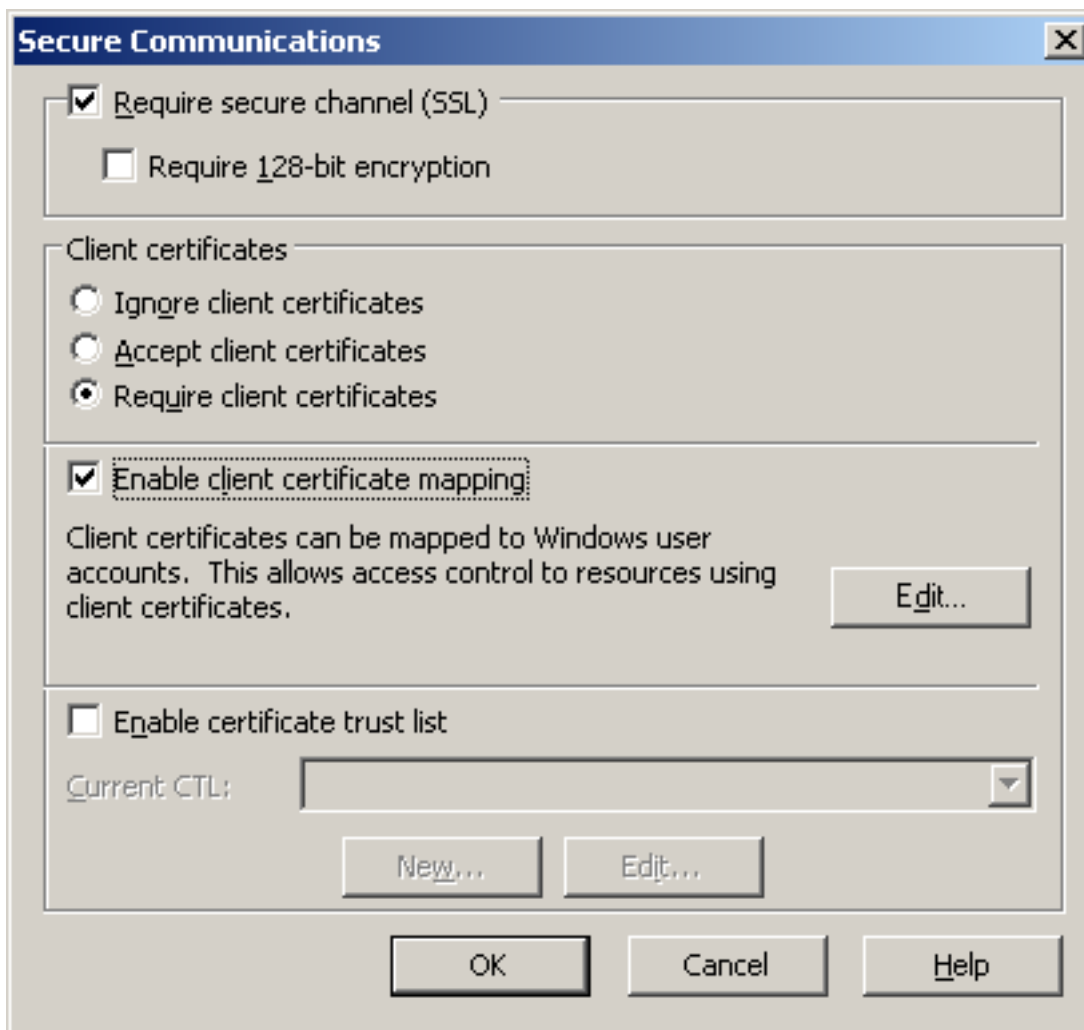


Figure 14-9: Enable Client Certificates in Secure Communications Directory Security Settings

Many-To-One Mappings

Many-to-one mappings are the easiest to configure because you can set it once and then forget it, provided you manage certificate chain trust settings properly on your IIS box and take precautions to issue only client certificates that will map properly to the Windows user account you intend. The impersonation that occurs when a user submits a many-to-one-mapped certificate to IIS controls the access privileges and permissions afforded to the server-side scripts and other logic implemented by a Web application. NTFS DACLs control everything else as though the mapped Windows user account were authenticated through a regular Windows network logon. The impersonation does not establish a logged-on-locally login. Figure 14-10 shows the many-to-one certificate mapping configuration window.

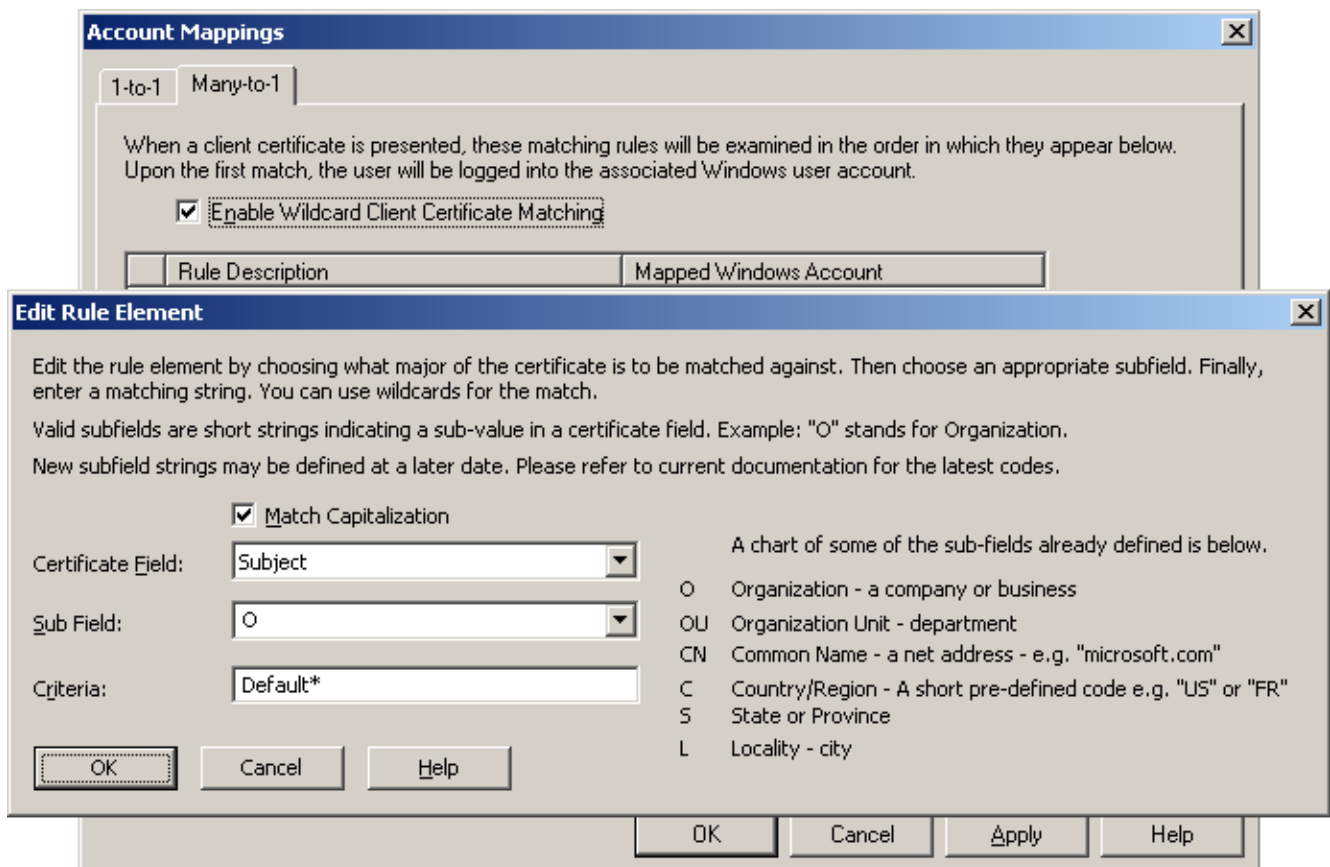


Figure 14-10: Map Many Certificates to a Single Windows User Account with Many-to-1

One-To-One Mappings

One-to-one mappings are the most secure but require the most effort to manage. Figure 14-11 shows a one-to-one mapping between a certificate with a particular Subject and Issuer and the Administrator Windows account. Click the Add button and have a certificate file handy because you can't add a one-to-one mapping without a copy of the actual certificate that a particular client will submit to authenticate with IIS and receive appropriate one-to-one account impersonation.

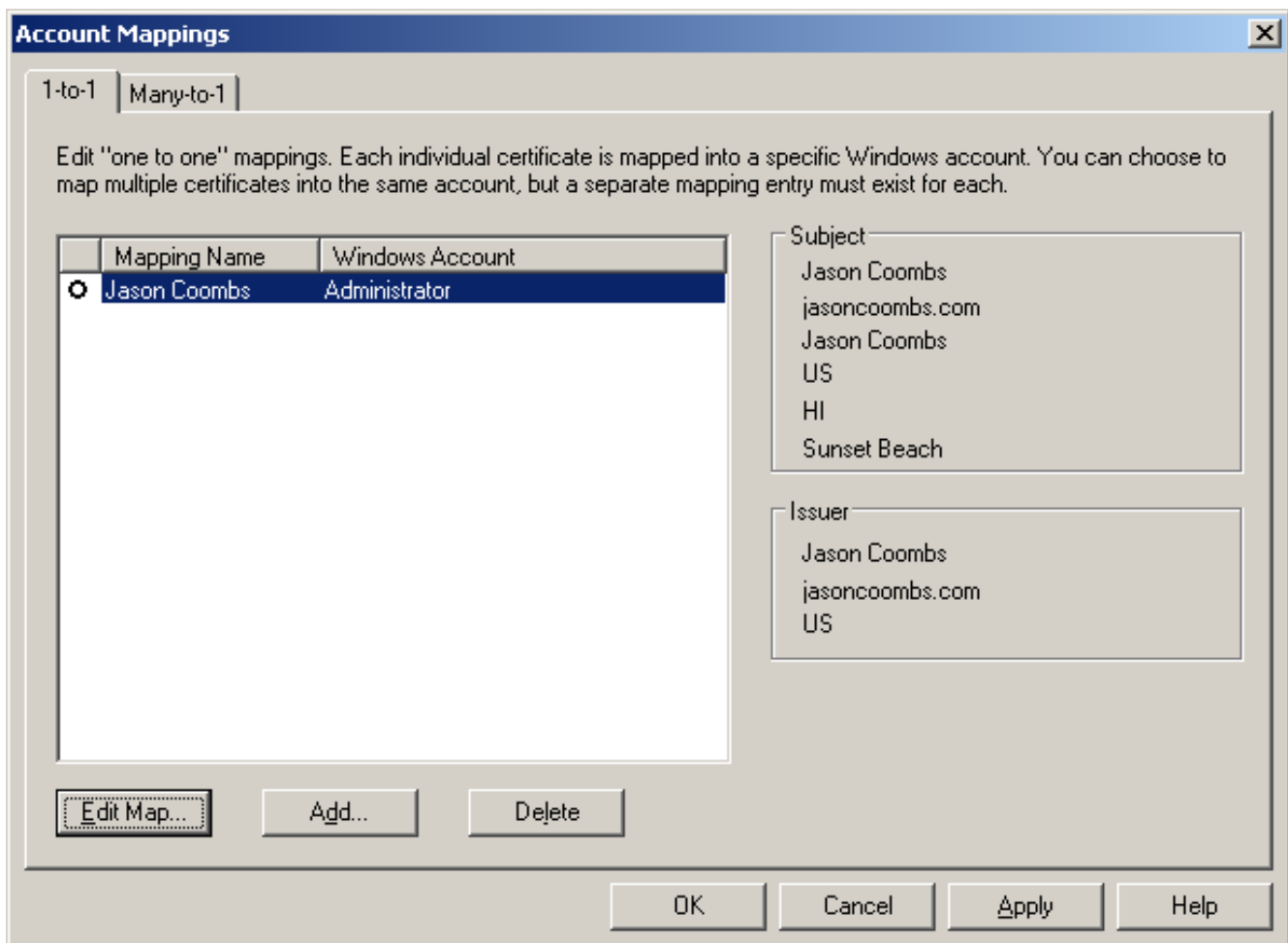


Figure 14-11: Associate a Single Windows User Account with a Single Client Certificate with 1-to-1

Secure Sockets Layer Encryption

Client Certificate Authentication requires Secure Sockets Layer (SSL) encryption in order to function because client and server authentication with X.509 PKI certificates is a feature of the SSL protocol. IIS rely on this feature of SSL to receive and verify client certificates, there is no other built-in support for client certificate authentication in IIS. It is technically possible, cryptographically, to achieve client and server authentication without encrypting communication itself. Verifiable digital signatures with Message Authentication Codes (MACs) make it unnecessary to encrypt communication except for the added feature of privacy. When Web applications use SSL client authentication, however, they automatically get data privacy with SSL encryption as well whether they want it or not. This isn't a bad thing unless the size of a user community is so large that concurrent access to shared server resources makes it difficult or prohibitively costly to load balance SSL encryption on every HTTP request. The solution to this problem is to SSL-encrypt only the initial authentication event that uses client certificates and from then on, for the duration of an authenticated session, relying on a session identifier such as an ASP.NET Forms Authentication token or Microsoft .NET Passport session ticket.

There are good reasons to require both client certificate authentication and conventional password authentication, especially when the alternative would result in automatic authentication. For example, client certificates are often associated with a particular computer used by a particular person, and as such any person who sits down at another person's computer after the person has logged in to the computer may be able to authenticate automatically against any service that recognizes the client certificate installed on that computer. This distinction between arbitrary possession of a private key and the ability to control a device that will automatically employ a private key is an important one to remember. An attacker who gains physical access to a computer for a short period of time may not have possession of that computer's private keys but for the duration of physical control over a computer an attacker can make use of private keys without appearing any different from the authentic owner of those keys. Requiring a password in addition to a client certificate solves this problem by forcing an attacker to capture a secret before they can control a device that has control of private keys. To take this practical protection to its logical conclusion you deploy password-protected smart cards that store a user's private keys so that an attacker must steal a physical card and capture a secret password to unlock access to its contents, then obtain physical access to a device that has corresponding client certificates installed or capture client certificates so that they can be installed on a different computer entirely for use in conjunction with the stolen smart card.

Configuring SSL in IIS is self-explanatory, and the Web Server Certificate Wizard shown in Figure 14-12 makes it possible to produce a new key pair and PKCS #10 CSR, as well as deliver it automatically to online CAs if any are accessible from your IIS box. Just follow the prompts after opening Web site properties in the MMC, selecting the Directory Security tab, and clicking on the Server Certificate button.



Figure 14-12: The Web Server Certificate Wizard Also Helps with Existing Certificate Installation

You don't need to rely on the Web Server Certificate Wizard to create a CSR and save it to file or send it to a CA for you if you have Certificate Services installed. Use Web Enrollment or a custom script to produce your own PKCS #10 CSR and key pair, then send the CSR to a CA of your choice. When your certificate is issued, the Wizard will install it for you from a file you select.

PKI certificates are extremely important for any IIS deployment that requires high security. Secure Sockets Layer encryption is the most common use for certificates in Web server deployments, but they are just one aspect of PKI. Certificates really do nothing more than bind a particular public key to a known entity, and certify that relationship with a chain of trust embodied in a series of digital signatures that stem from a root certification authority and optionally include intermediate and issuing CAs also. Certificate chains are designed in many PKI systems, and especially in Windows, to facilitate automated trust determinations to be made by interpreting arbitrary certificates based on the ability to verify each link in its trust chain. A whole host of problems arise from this design feature of Windows PKI that can have serious consequences if certificate chain trust policies are mismanaged or if software that makes use of certificate chains turns out to have unexpected bugs.

In addition to supporting SSL for data privacy through encryption, IIS also provides support for authentication of end-users with certificates. Client certificate authentication is powerful and versatile authentication option that avoids the common vulnerabilities inherent to password-based systems. However, unless smart cards are used to enforce end-user security policy and protection over access to a particular certified identity client certificates should not be a complete substitute for password-based credentials but rather should be supplementary.

Chapter 15: Publishing Points

Internet Information Services exist for the purpose of controlled dissemination of data and application services. IIS can't fulfill this purpose without preconfigured data sources or custom Web application code with which to service requests. Each input source of data or code that IIS rely on for request processing is called a publishing point. A single publishing point can easily serve content for any number of DNS domains depending upon IIS configuration settings that map physical directories, ISAPIs, and HTTP/1.1 Host Headers to distinct Web sites. Web application code deployed to a publishing point may also enable site-specific dynamic content creation based on variable request parameters.

Therefore a publishing point is a behind-the-scenes server-side concept of control and authority, not the client-side perception of a distinct Web site or application service. A single directory tree is often synonymous with a single publishing point, though there are no fixed rules or definition; any mechanism you deploy that has the effect of restricting authority and control over a source of data or code accessible through IIS constitutes a publishing point. Many different techniques and tools exist that connect IIS to data sources while enforcing access restrictions and controlling publishing. They all share common features such as the ability to authenticate authorized publishing requests, a system of selective content management, and vulnerabilities caused by too much complexity and not enough publishing point hardening. Picking a publishing point management solution that has few, if any, significant vulnerabilities isn't hard, it just requires a bit of common sense and some healthy skepticism of reinventing the wheel.

Optimal security is easy to achieve for any publishing point: simply disable all remote access to it and require an administrator to manually install new content using a local login to the box that is responsible for its content storage. Because this level of security isn't practical for many deployments of IIS, securing publishing points is a matter of managing the way that you allow new publishing points to be established, grant remote access to them, and prevent unanticipated problems that may arise when remote access solutions malfunction or become misconfigured. To begin with it's important to understand the differences between the built-in and the optional add-on integrated publishing point management features of IIS so that you can decide whether they meet your security and usage scenario requirements. Built-in IIS publishing features include File Transfer Protocol (FTP), Web Distributed Authoring and Versioning (WebDAV), and RFC 1867 HTTP file upload. Optional add-ons for publishing to IIS include the FrontPage Server Extensions, Background Intelligent Transfer Service (BITS), several third-party solutions, and conventional file copy or remote access tools like Point To Point Tunneling Protocol (PPTP) and Terminal Services where file transfer can occur through Network Neighborhood by copying files from a source location to a destination publishing point folder over a VPN.

Microsoft Knowledge Base Article Q253696 Cannot Access URL with ADO 2.5 and Internet Publishing Provider (MSDAIPP) on IIS 4.0 explains that there is no built-in support for either WebDAV or FrontPage Server Extensions in IIS 4.0.

Software tools designed to enable remote management of publishing points through built-in FTP or WebDAV features of IIS normally provide no scripting or programmability interfaces that would give you the ability to customize publishing procedure and implement additional security policies. As a result it is common to build your own tools for content deployment that combine access to a conventional password-protected publishing point, or a custom publishing interface based on RFC 1867 HTTP file upload or BITS, with custom code that implements additional security measures on both the client and the server. Such custom tools may be as simple as WSH scripts or they may be full client applications combined with Web application services. Where FTP and WebDAV are especially well-suited to the job of bulk file transfer, they provide no context-sensitive file management or data security features that should be added to the content deployment process before and after file copy operations are performed.

File Transfer Protocol Service

FTP is one of the most important services that enable remote management of a publishing point. The FTP protocol (RFC 959) is widely implemented and is supported by nearly every Web publishing tool that is capable of deploying files to a publishing point (with the notable exception of certain Microsoft Web publishing programs like FrontPage). In addition, an FTP service is often used as a publishing point of its own because of its built-in password-based authentication that by usage convention may allow anonymous access in addition to password-protected access by specific users. Most Web browsers in use today work just as well with ftp:// as with http:// URLs and will automatically login as the anonymous FTP user when an FTP server is configured to allow anonymous access. More common, however, is for FTP to be used only for password-protected remote management of files in a publishing point, with one or more Web site instances configured to look for content with which to service HTTP requests at the same physical directory tree used as the root directory of the FTP service. Application-specific request processing logic is easy to build into a Web site whereas FTP services are usually designed to provide generic abilities that can't be easily extended with new functionality and business logic. Lack of extensibility is a plus for security, and in many cases FTP will be the preferred method of remote management for a publishing point. However, FTP servers don't typically support any type of encryption, so all user credentials are transmitted in the clear over the wire and are subject to interception by eavesdroppers.

This is an easy risk to mitigate by simply establishing a different FTP user account for each publishing point and carefully preventing any FTP user account from being used for any other purpose than to authenticate with an FTP service. To further limit the damage that can possibly be done by compromised FTP user account credentials, you can implement a simple Web interface for content publishing approval where a different set of credentials, supplied via SSL to the Web application, must be entered in order to review pending content submitted via FTP for publication and approve or reject it. This makes it impossible for an attacker who does intercept FTP user account credentials to use them to publish arbitrary content and code to your IIS box. This two-stage publishing point management requiring SSL-encrypted credentials for publishing approval following unencrypted FTP content upload is easy to build and it is absolutely essential for proper hardening of your publishing points when they are managed with FTP. The advantages of FTP over all other publishing point remote management interfaces are numerous and

include the non-trivial benefit of security through maturity: FTP is a very early TCP/IP application protocol that has been examined in great detail for many years.

FTP, defined by RFC 959, can be found at <http://www.ietf.org/rfc/rfc0959.txt>

Most of the vulnerabilities present historically in IIS FTP are DoS conditions:

Q188348 Specially-Malformed FTP Requests May Create Denial of Service

Q189262 FTP Passive Mode May Terminate Session

Q293826 MS01-026: Pattern-Matching Function Can Cause Access Violation on FTP Server

Q317196 MS02-018: Patch Available for Denial of Service Through FTP Status Request Vulnerability

In addition to lack of encryption support, the FTP service in IIS wasn't designed optimally for enabling many different content authors to control their own publishing points without being able to also see or control publishing points that belong to other users all on the same server box. Worst of all, because NTFS DACLs were the only mechanism available to restrict access to FTP directories and files, it was often the case that DACLs would get modified in a publishing point in such a way that other users or even the anonymous user would be able to access, and write to, files or folders through FTP that they weren't supposed to be able to view or edit. Because DACLs are cumbersome to analyze quickly, requiring mouse clicks and popup windows reviewed separately for every file and folder, administrative oversight of DACLs as the sole security measure is terribly inadequate and many IIS deployments are forced to use a different FTP server as a result. Prior to IIS 6, the FTP service provided in IIS just wasn't optimal. Even creating separate FTP server instances for each publishing point wasn't a solid solution to these problems because any user account valid for one IIS FTP server would also be valid for all of the others, and once again DACLs were the only protection against unwanted access or even unauthorized publishing. Finally, with IIS 6, Microsoft has resolved these difficulties.

IIS 6 FTP User Isolation Mode

Under IIS 6 each FTP server instance now has the option of restricting a logged-in user to their own physical directory with a feature known as user isolation. Figure 15-1 shows how to activate FTP user isolation mode where a directory beneath the root of the FTP server instance automatically becomes the apparent root directory of any authenticated user. Each user is given a different isolated root directory by creating a directory beneath the FTP server instance root with a name that matches the user ID. Integration with Active Directory enables users whose directory service user object contains the optional FTProot and FTPdir attributes to login with a full UNC path to their isolated FTP directory provided dynamically instead of based on user ID.

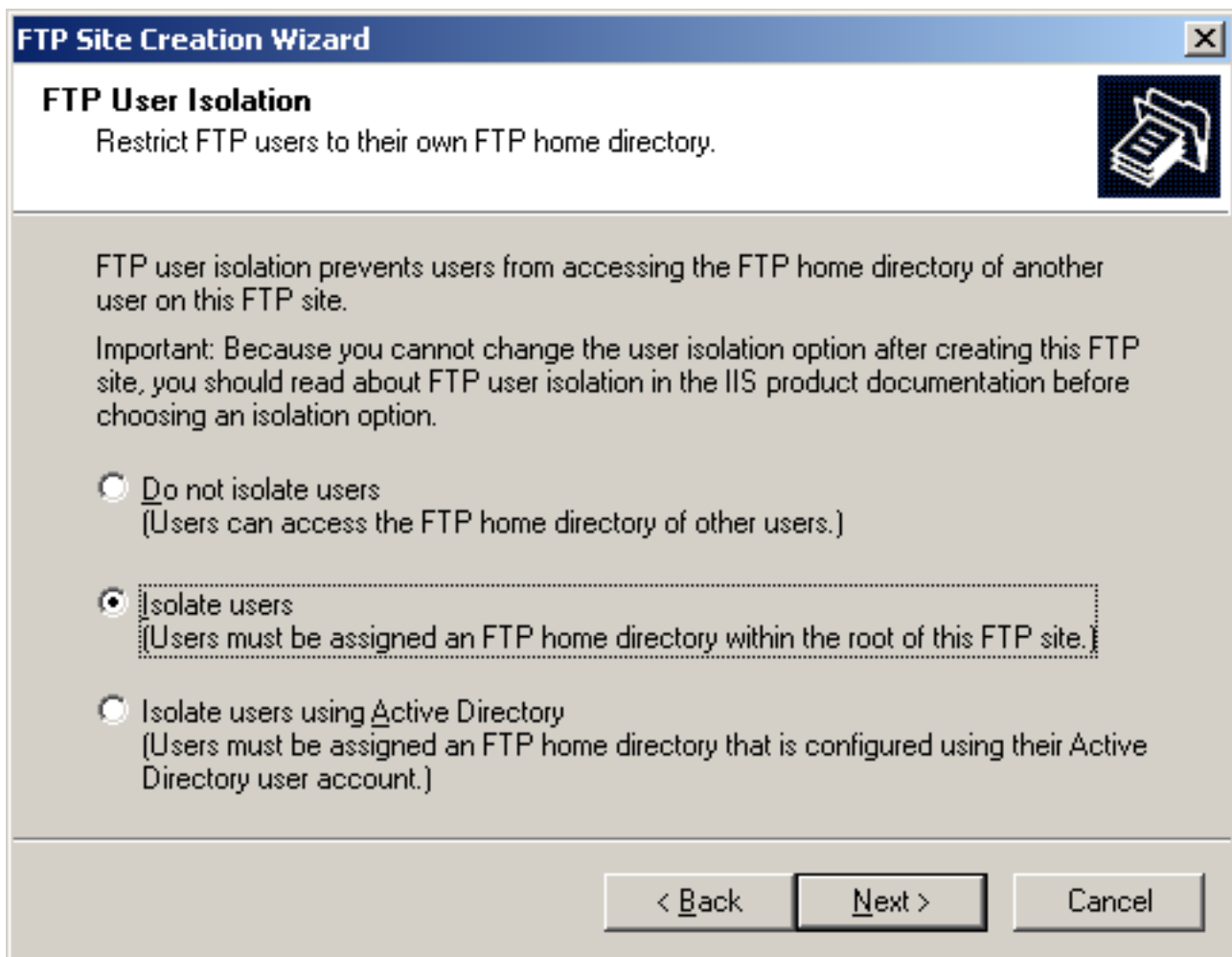


Figure 15-1: FTP User Isolation Enforces Access Restriction Beyond DACLs

Isolated user directories are created beneath a subdirectory of the FTP root directory that corresponds to the Windows domain of the user account or LocalUser if the account exists as a local user account on the IIS box. When users authenticate with the FTP service they can optionally provide user ID in the form domain\user where domain is the Windows domain and user is the full user ID recognized in the domain. For a domain user, when FTP user isolation mode is enabled but Active Directory is not, the full path to the user's isolated FTP directory looks like this:

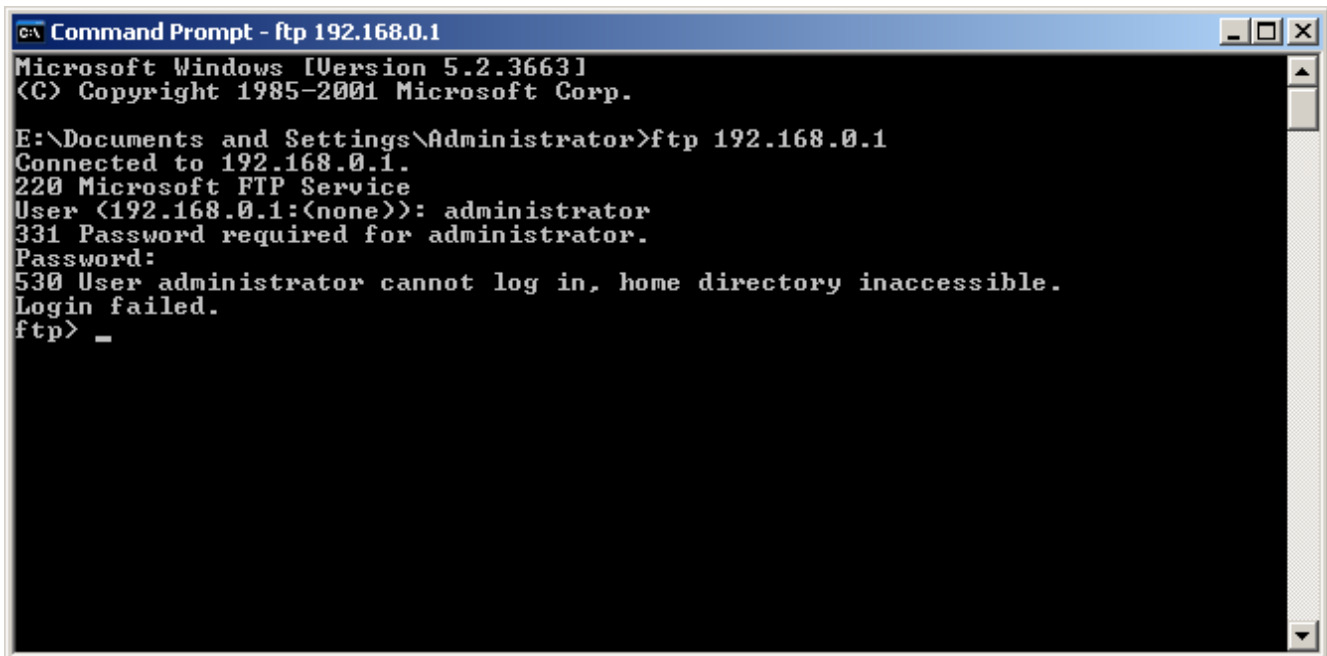
```
C:\inetpub\ftproot\domain\user
```

User accounts authenticated against the local users database rather than any Windows domain have isolated FTP directories under LocalUser instead of domain. For anonymous FTP, a directory named Public is required beneath the LocalUser subdirectory as shown here:

```
C:\inetpub\ftproot\LocalUser\Public
```

FTP user isolation mode makes it possible to rely on password protection with Windows user accounts and DACLs for imposing access restrictions without having to worry about developing vulnerabilities over time if DACLs are inadvertently or intentionally changed.

Rather than being completely dependent on perfect DACLs, IIS will now prevent users from even attempting to access other users' FTP home directories. This is more consistent with the least privilege principle that says user accounts should always operate with only the minimum set of access rights and abilities required to carry out necessary and authorized activities. As shown in Figure 15-2, any user account that authenticates with an FTP server operating in user isolation mode will be denied access unless its isolated home directory exists and is accessible in the user's local security context, even the built-in administrator account.



```
Microsoft Windows [Version 5.2.3663]
(C) Copyright 1985-2001 Microsoft Corp.

E:\Documents and Settings\Administrator>ftp 192.168.0.1
Connected to 192.168.0.1.
220 Microsoft FTP Service
User (192.168.0.1:(none)): administrator
331 Password required for administrator.
Password:
530 User administrator cannot log in, home directory inaccessible.
Login failed.
ftp> _
```

Figure 15-2: A User Account Without an Isolated Home Directory is Denied Access

Although FTP uses no encryption and passes credentials in the clear, this only results in compromised credentials if there is actually an eavesdropper capturing network traffic between client and server. The effect of this credential capture is unauthorized access to a publishing point, which may be harmless with respect to access to published files if the files are accessible by anonymous Web site users anyway. If only static content is allowed on the server, with no scripting or server-side include capabilities enabled, then Web site defacement, cross-site scripting (XSS), and denial of service (DoS) resulting from content deletion and perhaps hard disk resource consumption or bandwidth consumption through warez swapping or other file trading activity are the potential damages. All of these harmful events can be detected and shutdown very quickly simply by changing the compromised FTP user password and restoring original files from backup. With such a simple incident response possible and threat containment built-in to the design of the FTP service, there are many instances where the risk of unencrypted FTP is reasonable.

Ideally no credentials would ever be compromised. But encryption doesn't prevent all credential theft, it simply reduces the likelihood of such theft by way of eavesdropping. What you really need, more than encryption, is an optimal security policy for deployment of updates to your publishing points and this requires at least two stages. The first acts as a staging area for final quality assurance and approval prior to publishing, and the second

stage involves moving approved content into production where an update to existing content, or new content, is deployed for access by end-users and site visitors. As long as different credentials are required for each stage, theft of first-stage credentials will result at worst in disk space DoS and pirated warez, digital movies, and digital music swapping until you detect the spike in bandwidth usage or problems caused by inadequate available disk space. Theft of second-stage credentials will be essentially useless to attackers.

Changing FTP passwords frequently (as often as practical, and more often for more important publishing points) and keeping FTP paths as dynamic and unpredictable as possible are far more valuable for defending against unauthorized access to a publishing point than is encryption alone. And there's no reason you couldn't deploy encryption and digital signatures for published files as part of your security policy. The second stage can easily include decryption and digital signature verification of each uploaded file, leaving only threat of eavesdropping credential theft and any inconvenience it causes.

Using Hidden FTP Virtual Directories

A defensive configuration technique worthy of note whether or not you're using IIS 6 is the creation of virtual directories under the root of an FTP server instance. When a DACL mistake is made on files or folders accessible only through an FTP virtual directory the vulnerability is not immediately exploitable by anyone who happens to gain access to the FTP server root with valid user credentials. This is because virtual directories don't show up in the file and directory listing, they can only be accessed with an explicit change working directory command or an ftp:// URL that points at the full path of the item whose DACL is insecure. Figure 15-3 shows a simple FTP virtual directory and the resulting directory listing when a user authorized to login to the FTP server root requests a list of the root directory contents.

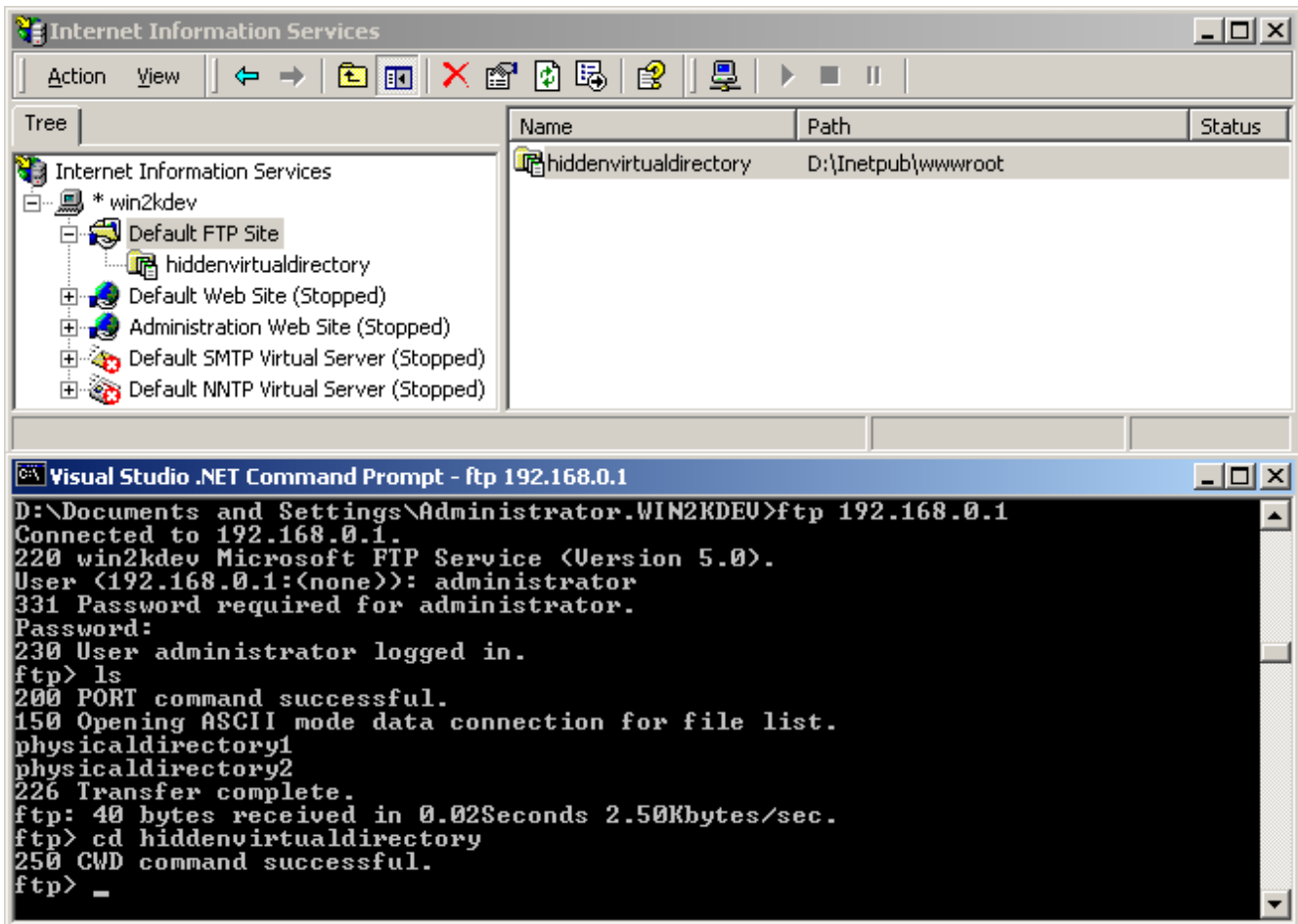


Figure 15-3: Virtual Directories Are a Common Defense Against DACL Problems with IIS FTP

The hidden virtual directory can have a lengthy directory name that does not correspond to the name of its physical directory. A long virtual directory name that conforms to strong password rules (random letters and numbers, with uppercase and lowercase) is just as good as a password because IIS FTP will not reveal the existence of virtual directories to client programs, therefore an attacker would have to guess a valid directory name in order to gain access to an FTP virtual directory. Because virtual directory names are stored in the metabase under each FTP server instance `IIsFtpServer` key (`MSFTSPVC/N`) as separate `IIsFtpVirtualDir` object values, it isn't difficult to use Active Directory Services Interface (ADSI) to script the automated modification of virtual directories' metabase entries so that even eavesdropping to intercept a hidden virtual directory path isn't enough for an attacker to compromise all publishing to the publishing point indefinitely. You could give your users a one-time use password list and a way to login and logout of publishing point management mode using a Web application so that each FTP session uses a different virtual directory name. When the user wishes to login via FTP to manage a publishing point, they would indicate as much to your Web application which would then set the name of the virtual directory to the next password in the user's predetermined one-time use password list. When the user logs out (or after an application-specific timeout period) the virtual directory can be removed entirely to prevent access to the publishing point.

WebDAV Distributed Authoring and Versioning

WebDAV is Microsoft's alternative to FTP for remote management of publishing points. Its most important benefit over FTP is the fact that it is HTTP-based and therefore can benefit from the use of SSL encryption and offer more options for authentication and custom publishing point administration behaviors on the server. WebDAV is also designed to allow file locking to prevent multiple authors from overwriting each other's work when producing content collaboratively. These and other features of WebDAV may make it desirable, but you must carefully weigh the benefit versus the increased risk of exposing WebDAV functionality on publishing points. Known problems in the past with WebDAV, and its technical design, suggest the same type of architectural security flaws that plagued ISAPI and IIS mechanisms that were prone to vulnerabilities because of excessive complexity and too much code injected into request processing making the architecture very difficult to harden. FTP, on the other hand, has had relatively few security vulnerabilities and one of the very troubling things about WebDAV is that Microsoft is continuing to add new features to it. This makes the whole interface a moving target for security hardening, and there is just no way to tell what new vulnerabilities your IIS box might be exposed to with each hotfix or service pack that provides a new build of WebDAV support. WebDAV also includes some extremely complex features for performing searches and other administrative tasks using XML which are prime candidates for buffer overflows and inadequately-tested and inadequately-security hardened code as a result of the difficulty anyone would have performing software quality assurance on a system with too many usage combinations and too many possible paths through the code.

See the following Knowledge Base articles on known WebDAV security issues:

- Q291845 Malformed WebDAV Request Can Cause IIS to Exhaust CPU Resource
- Q296441 MS01-022: WebDAV Service Provider Can Allow Scripts to Levy Requests as a User
- Q298340 MS01-044: Patch Available for WebDAV Denial of Service
- Q307934 Locking Down WebDAV Through ACL Still Allows PUT & DELETE

WebDAV is an open standard defined by RFC 2518 entitled "HTTP Extensions for Distributed Authoring – WEBDAV" and RFC 3253 entitled "Versioning Extensions to WebDAV (Web Distributed Authoring and Versioning)" that Microsoft participated in designing as part of an IETF working group. The protocol itself is unlikely to be found to have security flaws because there are no cryptographic or security-oriented algorithms specified by WebDAV RFCs. But flawed protocols are rarely the source of security problems in the first place, and there is no doubt that implementations of WebDAV do have and will continue to have bugs that impact security. The additional auditing features that are enabled thanks to WebDAV's version control mechanisms can partially offset some of this risk assuming that a particular vulnerability doesn't result in arbitrary modifications to versioning history data, but this doesn't change the fact that WebDAV is a protocol designed to do something that is a fundamental security risk: increase the ease with which a remote client can modify resources on a publishing point. It should not be easy to modify resources located on servers. The following excerpts of RFC 3253 and 2518 give a clear picture of how WebDAV itself is a new potential vulnerability:

RFC 3253 Section 16.3 Security Through Obscurity

While it is acknowledged that "obscurity" is not an effective means of security, it is often a good technique to keep honest people honest. Within this protocol, version URLs, version history URLs, and working resource URLs are generated by the server and can be properly obfuscated so as not to draw attention to them. For example, a version of "http://foobar.com/reviews/salaries.html" might be assigned a URL such as "http://foobar.com/repo/4934943".

RFC 2518 Section 17.3 Security through Obscurity

WebDAV provides, through the PROPFIND method, a mechanism for listing the member resources of a collection. This greatly diminishes the effectiveness of security or privacy techniques that rely only on the difficulty of discovering the names of network resources. Users of WebDAV servers are encouraged to use access control techniques to prevent unwanted access to resources, rather than depending on the relative obscurity of their resource names.

RFC 3253 can be found online at <http://www.ietf.org/rfc/rfc3253.txt>

RFC 2518 is available online at <http://www.ietf.org/rfc/rfc2518.txt>

As you can see, the different RFC authors appear to contradict each other on the point of security which they incorrectly label "security through obscurity". In RFC 2518 it is stated that security through obfuscated file or path names is likely made useless by the very search capabilities defined by the protocol. Because the authors of RFC 3253 misunderstood what the authors of RFC 2518 defined as "security through obscurity" (recall that this term properly used means that an adversary is unable to mount an attack without first discovering the details of a secret algorithm or proprietary system design and should not be applied to little bits of secret information like passwords or hidden directory paths) statements directly counter to the intent of Section 17.3 in RFC 2518 were added to Section 16.3 of RFC 3253. The bottom line is that this type of contradiction is commonplace in RFC documents and other protocol standards, as software developers and system engineers focus on features rather than real-world information security consequences. Some of the features of WebDAV don't have easy alternative implementation paths within the confines of a more mature remote publishing protocol like FTP, and certain WebDAV features are potentially valuable, but not so much so that a security-conscious administrator would intentionally expose any WebDAV-compliant publishing point on any network where attackers are likely to lurk, even with the protections of SSL and authentication countermeasures.

Disabling WebDAV for Your Entire IIS Box

By default, every Web site and folder are configured to respond to WebDAV requests in IIS 5 whether or not publishing points are configured for WebDAV access based on specific authentication credentials, file and folder DACLS, and metabase settings. There are probably still vulnerabilities left undiscovered in Microsoft's WebDAV implementation that can be exploited remotely by an attacker, and this is by far a more serious concern than the potential for eavesdropping on publishing point management. Especially considering that there is rarely any point in protecting content that is deployed to IIS from being intercepted as it is published since the same content is often accessible by anonymous

Web users once it has been published successfully. FTP with IIS 6 user isolation mode or hidden virtual directories presents a certain amount of risk of compromised publishing credentials, but this is far less risky than allowing WebDAV and other complicated and potentially-vulnerable server-side code to respond to every publishing point management request. A buffer overflow exploit of WebDAV that results in complete system compromise is not a reasonable trade-off of risk compared to FTP, and WebDAV has the serious technical drawback of being designed to access published content in-place rather than easily facilitating a two-stage deployment architecture where approval can be required to authorize the publication of pending changes to Web site content.

WebDAV is implemented in HTTPEXT.DLL, which is always installed as part of the core IIS 5 platform. As part of Windows 2000 Security Rollup Package 1, a Registry value under W3SVC\Parameters key was created that enables WebDAV to be disabled entirely for the whole IIS box and every Web server instance hosted on it. Set the following DWORD value to 1 if you wish to disable WebDAV completely:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\
W3SVC\Parameters\DisableWebDAV
```

WebDAV is an optional feature for IIS 6 that is disabled by default. To enable it, you must explicitly allow the WebDAV ISAPI extension (httpext.dll) and you can do so easily with the IIS MMC. The DisableWebDAV Registry value is unnecessary.

FrontPage Server Extensions (FPSE)

Before WebDAV emerged as the standards-based HTTP 1.1 extensions for remote publishing point administration Microsoft developed the FrontPage Server Extensions (FPSE) as an HTTP-based alternative to FTP. Like WebDAV, FPSE goes overboard with features instead of simply providing an SSL-enabled HTTP-based replacement for FTP. Unlike WebDAV, FrontPage attempts to provide common Web application features as modular components that novice Web developers can use to build more interactive site content without the burdensome necessity of acquiring coding skills. Little more than these basic facts are required in order to understand that FPSE pose a substantial security risk in exchange for very little real benefit. This is one reason that WebDAV has been created as its replacement. While both WebDAV and FPSE rely on DACLs as the core protection against unauthorized access to files, they can both be used in conjunction with SSL and client certificate authentication which can make them more secure with respect to eavesdropping and credential theft versus FTP.

DACL misconfiguration and authentication settings errors are common in FPSE-enabled servers. Often the harm that is done as a result of these problems is limited to bandwidth theft when a FPSE publishing point is hijacked for file trading by copyright pirates. Historically FPSE has exposed severe security flaws including buffer overflow vulnerabilities that could be exploited remotely by an attacker for the purpose of executing arbitrary malicious code on the server. Because of the design of FPSE and its intended purpose, it is likely that there are more security bugs left to be discovered and fixed. FPSE should never be used now that WebDAV is available as a viable alternative. If you must use one of these two features because of a need to enable Web Folders or FrontPage-style access to publishing points or because of a desire to SSL-encrypt

publishing operations and credentials you should always use WebDAV rather than FPSE. Note also that the Office 2000 Server Extensions, a distinct product install that provides a superset of FPSE, should likewise be avoided.

See Microsoft Knowledge Base Article Q235027 Differences Between Office 2000 Server Extensions and FrontPage 2000 Server Extensions

Microsoft Knowledge Base Article Q260267 FrontPage Server Extensions Image Map Files Expose Security Vulnerability describes buffer overflow vulnerabilities in both server-side image map files shipped with FPSE 97 and 98, htmimage.exe and imagemap.exe.

It is possible that the FrontPage client program will force either of these CGI executables to be installed on the server without informing either the user or server administrator that it is doing so. To defend against this flaw in FrontPage client, FPSE 2000 denies uploading of files to folders marked as executable by default. This prevents FrontPage from successfully deploying any security-vulnerable CGI executable to a CGI-capable folder on the server.

See Frequently Asked Questions: Microsoft Security Bulletin MS00-028
<http://www.microsoft.com/technet/security/bulletin/fq00-028.asp>

FPSE include an optional feature called Visual Studio RAD (Remote Application Deployment) that enables authorized developers to modify COM+ settings as well as install and register COM+ components remotely. Although this RAD facility is normally not installed on production servers, it's important to note that it contains a buffer overflow vulnerability first fixed in Windows 2000 Security Rollup Package 1 (SRP 1) in January, 2002. Like the buffer overflow vulnerabilities in server-side image map CGIs installed by force by the FrontPage client, there is a distinct possibility that a Visual Interdev developer may cause the old, vulnerable version of RAD to be deployed to an IIS box in place of the updated SRP 1 version if the developer is given administrative access to the box and not warned of this threat.

See MS Knowledge Base Article Q300477 FP2000: MS01-035: Potential Buffer Overrun Vulnerability in Visual Studio RAD (Remote Application Deployment)

FPSE 2000 Service Release 1.3 (SR 1.3) includes a fix for a buffer overflow vulnerability in FP4aReg.dll that allows a long argument to overflow a stack buffer. See KB Article Q306118 FPSE2000: List of Issues Fixed in FPSE SR 1.3

One of the unfortunate ironies of FPSE's legacy is that prior to version 2000 of the FrontPage client and with all versions of Visual Interdev, support for SSL was severely limited due to hard-coded root CA certificates. Only four CAs could issue SSL certificates that would be trusted by most clients of FPSE-enabled publishing points. Two of AT&T's root CAs and two of VeriSign's root CAs were the only ones trusted prior to FrontPage 2000 when a switch to reliance on the WinInet API created the opposite security problem: any root CA certificate trusted by Internet Explorer on the Windows platform is now automatically trusted by FrontPage 2000. Further, prior to FrontPage 2000, SSL used with FPSE was limited to 40-bit low encryption mode by these same hard-coded facilities that were meant to replace FTP with an encrypted interface for remote publishing point

management. FPSE could be described as a comedy of errors if not for the sad fact that a standards-based solution for secure remote publishing point management under Windows would have avoided numerous critical vulnerabilities in IIS boxes that instead adopted FPSE and found themselves to be vulnerable as a result. Considering that the only meaningful benefit of FPSE over FTP was the availability of SSL for more secure publishing, the incompatibility of FrontPage/Visual Interdev with real-world SSL is disturbing even in retrospect.

Microsoft Knowledge Base Article Q267999 FIX: FrontPage 98 Clients and Visual InterDev Cannot Open Web Site Through SSL explains that many FrontPage and Visual Interdev users experienced a sudden DoS condition as a result of replacement of a common VeriSign root CA certificate for 40-bit SSL.

Once FPSE was deployed widely, client interfaces to its exposed publishing points also became a moving target. The Web Extender Client (WEC) is a component included in Office 2000, Windows 2000, and Windows Millennium that enables the Web Folders feature for Windows Explorer-style publishing point management. As originally designed, WEC did not pay attention to the security settings established by Internet Options and therefore WEC viewed the entire world essentially as part of the local Intranet zone. Any WEC-enabled Windows box would attempt to automatically authenticate using NTLM with any server that supplied an NTLM challenge. Because of the well-known security flaw caused by weak NTLM hashes and the availability of cracking tools that could easily discover a plaintext password when given its NTLM hash, simply browsing the Web using any WEC-enabled Windows box that has not been patched could result in credential theft.

For more information about the Web Client Security Update for Office 2000, please browse to the following Microsoft Security Bulletin:

<http://www.microsoft.com/technet/security/bulletin/ms01-001.asp>

This problem was first corrected in Windows 2000 Service Pack 2 and the Web Client Security Update for Office 2000 in which the file fp4awec.dll was updated to version 4.0.2.4715.

Publishing Point IIS Admin Object ADSI Scripts

Provisioning new Web sites with publishing points that supply data and code to them can be a complicated process when there are many configuration changes required for each in order to achieve a desired state of security hardening balanced against feature and flexibility requirements. Each of your Web sites may have an IDS layer, a class framework for application development, and a custom logging ISAPI that must be set up properly in addition to typical settings such as authentication options and DACLs. Automating these provisioning steps with code can be accomplished easily thanks to the existence of the IIS Admin Objects that expose Automation-compliant interfaces.

Automating Web Site Creation

The key to providing cost-effective Web hosting service is automating routine tasks. Whether your company is a Web Presence Provider (WPP) or is just large enough to have its own

Internet department, cost savings is but one benefit of the ability to fully automate Web site creation. When combined with real-time credit card payment processing you realize the potential of self-service automated Web hosting. It gives you the ability to automate the setup of additional servers in your server farm. You can take advantage of WebDAV or the ability to automate FTP service provisioning to support on-demand creation of Web Folders for managed remote file storage. Or you can just free up resources to tackle more important tasks by scripting the repetitive steps required for deployment of Web-based application services and sites. When new Web sites are created by executing code that will always remember to properly harden everything and configure it according to your security requirements you can also prevent unnecessary human error in such a repetitive fundamental task.

IIS are designed to be configurable through scripts that use Active Directory Services Interface (ADSI) to modify the Metabase. Inside the Metabase are stored all of the settings that control the operation and configuration of IIS, including the definition of specific Web sites and Web applications. In addition, IIS add-ons like Site Server's Personalization & Membership store configuration inside the Metabase. This makes it possible for ADSI scripts to custom tailor the settings of every aspect of IIS and its ancillary services, including all security preferences and settings.

Scripting New Web Site Creation

Creating a new Web site with Visual Basic Script can be accomplished using either Windows Script Host (WSH) or Active Server Pages (ASP) environments. Only a few lines of code are required to create a new Web site due to the inheritance model of the Metabase structure. As in the Windows Registry, the Metabase consists of containers, called keys, and values, called properties, stored inside those keys. Keys can exist inside other keys and alongside properties, thus producing a logical data hierarchy. Unlike the Registry, the Metabase uses a class structure where keys can be a particular KeyType, or class type, and inherit property values from keys higher up in the tree. Properties that are not explicitly set in a key can be inherited from keys of the same or different KeyType if the inherit flag is set for the property in one of the parent keys.

Additionally, ADSI provides an object extension model whereby classes of a particular type can be given special functionality not present in the underlying data directory. Through this extension model, Microsoft has built and provides as part of IIS a set of COM objects known as the IIS Admin Objects. Whenever ADSI is used to access a key from the IIS Metabase, an instance of the IIS Admin Object is created that corresponds to the KeyType of the Metabase key. Methods of the IIS Admin Object can be used to make changes to the Metabase or perform other actions that are beyond the original design of the Metabase. Nothing special need be done to instantiate one of the IIS Admin Objects, simply binding to a key in the IIS Metabase using ADSI triggers the COM object instantiation for you. The following example script shows how ADSI is used to bind to the IIS Metabase key for the root IIsWebService IIS Admin Object so that a new IIsWebServer instance can be created.

```
Set ServiceObj = GetObject("IIS://localhost/W3SVC")
Set ServerObj = ServiceObj.Create("IIsWebServer","100")
ServerObj.ServerSize = 1 ' Medium: 10k to 100k hits/day
```

```
ServerObj.ServerComment = "New Web Server Instance"  
ServerObj.ServerBindings = ":80:"  
ServerObj.SetInfo
```

GetObject is the normal Visual Basic function call used to instantiate a COM object. In this case the COM object being instantiated is an ADSI object bound to the IIS Metabase at the address shown. ADSI automatically determines the KeyType of the Metabase key and returns an object instance that implements additional COM interfaces if appropriate. Since the object being bound to in the Metabase is of KeyType IIsWebService, the object returned by ADSI implements the IIsWebService IIS Admin Object interface in addition to the standard ADSI interfaces. The Create method is a standard ADSI object interface method used to create a new key in the Metabase. The first parameter to Create is the class type (KeyType) of the key to be created and the second parameter is the name to assign to the new key. IIS prefer to name Web server instances with unique numbers, so this particular Create will succeed only if there is not already a Web server instance ID 100 present on the system. The instance ID will normally need to be determined at runtime rather than being hardcoded as in this example. You might attempt object creation in a loop that increments instance ID until the SetInfo call completes without an error or use some other methodology to determine the next unused instance ID. There is no need for instance ID to be assigned sequentially, so a random walk algorithm works also.

ServerBindings indicate to IIS which IP address(es) and port number to bind to for the new Web server instance as well as optionally specifying the HTTP 1.1 Host Headers, if any, that will be required in order for the new Web site to receive the incoming HTTP request. Host Header values are necessary whenever software virtual hosting is used on your IIS box to allow multiple Web sites to share the same IP address, but it is a very good idea to require it at all times in order to prevent access to any IIS-hosted site by HTTP clients that aren't HTTP/1.1-compliant. Several exploits exist that take advantage of the lack of Host Headers in HTTP/1.0 to facilitate data theft and certain XSS attacks, so by default all HTTP/1.0 clients should be given an error message that instructs them to upgrade if they wish to access one of your Web sites. This is essential in order to protect the end-user from themselves, as old HTTP/1.0 Web browsers are usually plagued by other security problems as well. SetInfo is an ADSI interface method that commits to the underlying directory (in this case the Metabase) any changes made in memory to property values.

IIsWebServer is the KeyType of any Web server instance in IIS. However, by itself it does not completely define a Web server instance. Another KeyType, IIsWebVirtualDir, must be added as a child of the IIsWebServer key to complete the site configuration. This additional key is necessary in order to tell IIS where to find the physical root directory, and the key must be named "Root". The following script shows the completion of initial set up for the new Web server instance. The Path attribute is the full path to the directory on the hard drive that will serve as the root directory for the site. You will most likely determine the path at runtime and create a new folder using Visual Basic Script's Scripting.FileSystemObject or something like it. This example shows a predetermined path for simplicity.

```
Set VDirObj = ServerObj.Create("IIsWebVirtualDir","Root")
```

```
VDirObj.Path = "C:\inetpub\wwwroot"  
VDirObj.AccessRead = True  
VDirObj.AccessWrite = False  
VDirObj.AccessScript = True  
VDirObj.SetInfo  
VDirObj.AppCreate True  
ServerObj.Start
```

The new `IIsWebVirtualDir` named "Root", which acts as the root directory for the Web server instance, must be defined as a Web application in order for many of the features of Active Server Pages to work properly. The `AppCreate` method causes such a definition to be added to the Metabase. This is an example of an ADSI Extension method provided by an IIS Admin COM object that extends the functionality of ADSI to simplify the configuration of the underlying data directory. With one method invocation, whatever IIS needs to have done in order to properly configure a new Web application is taken care of automatically. Another example is the `Start` method, which causes IIS to start the new Web server instance so that browsers can start sending it requests. It is interesting to note that any programmer can create ADSI Extensions to build applications around ADSI. For more information about this topic, look for documentation of the `IADsExtension` interface or search on "ADSI Extensions" in the MSDN Library.

The preceding steps are implemented for you by a vbscript utility installed along with IIS version 5 in the `AdminScripts` directory beneath `inetpub`. The utility named `mkw3site.vbs` can be used from the command prompt to create new `IIsWebServer` object and its required `IIsWebVirtualDir` `Root` in the Metabase. As long as you are logged in as Administrator when you invoke `mkw3site.vbs` you'll be able to use it to add new Web sites. If you wish to implement these steps from within ASP script, you must first configure permissions on the Metabase to enable editing by the Windows user account that will be active in the context of ASP script interpretation.

Configuring Metabase Permissions

If you type the lines of script shown in the previous two samples into a text editor and save them to a file with a `.vbs` file extension, you can invoke the script in the Windows Script Host. However, the script will be unable to modify the Metabase if the user account used to login to Windows does not have Administrator privileges. This is because the Metabase has an Access Control List associated with each key that contains Access Control Entries for each user or group that has permission to access the Metabase key. A utility provided by Microsoft called "MetaAcl" can be used to add, modify, or delete the ACE for a user or group in the ACL for any Metabase key. `MetaAcl` is distributed as a `.vbs` file named `metaacl.vbs` and must be installed manually prior to IIS 6. It was first provided with one of the NT Option Pack add-ons. Search the MSDN Web site to find and download the latest version. As of this writing, knowledge base article Q267904 contained a link to download the `MetaAcl` utility. A related ADSI utility available from Microsoft is called `adsutil.vbs`.

`MetaAcl` can either display existing ACL settings or modify them. To display the ACL for a Metabase key, go to the command line and change directory to wherever you installed the `MetaAcl` script then type "metaacl" followed by the path to an existing Metabase key

enclosed within double quotes. Each of the ACEs in the ACL is displayed iteratively in individual message box pop-ups. Not the most user friendly of utilities, but it gets the job done. To add or modify ACEs, you specify the user or group name as a second parameter followed by a third parameter listing the permissions to grant from Table 15-1. For example, the following line adds the right for the user NTDOMAIN\User to read, write, and enumerate everything under the root IIS Metabase key but does not give permission to modify ACL settings.

```
MetaAcl "IIS://Localhost/W3SVC" NTDOMAIN\User RWSUE
```

Table 15-1: Metaacl accepts any combination of six permissions settings.

Permission	Description
R	Read
W	Write
S	Restricted write
U	Unsecure properties read
E	Enumerate keys
D	Write DACL (permissions)

In order to modify the Metabase using ADSI from within Active Server Pages script, you must first use Metaacl to grant the appropriate permissions to the server's impersonation account under which the actions initiated by the hosting IIS Web server instance are performed. Although granting Metabase permissions to an IIS impersonation account does create a potential security risk for your server, the risk is often minimal because in order to exploit such Metabase permissions a hacker must be able to write script code and get it interpreted on your server. If a hacker is able to do this, then you have many other problems to worry about besides the fact that the hacker can read and edit your Metabase.

A good two-stage publishing authorization procedure is the best defense against unauthorized script execution on your IIS box.

The benefits offered by the ability to modify the Metabase from within server-side scripts are often worth the extra work to make sure your IIS box is secure so that your scripts are never modified by an intruder. Granting Metabase permissions is not part of automated site setup and only needs to be done once manually when you first deploy scripts to a Web server instance where automated site setup will be initiated. If you operate a hosting business where third parties can write server-side scripts that will be interpreted on your IIS box within the context of a hosted Web site, you must consider your clients to be hackers or potential hackers. This means that you cannot allow hosted Web sites to use the impersonation account that you have granted permission to modify the Metabase. Any Web server instance that hosts scripts that automate modification of the Metabase can and should have its own impersonation account. You will normally leave the impersonation account for regular Web sites not used in the automated modification of the Metabase set to the default Master Properties' setting for the WWW Service. Master Properties for IIS can be viewed and set by opening the Properties window for the server node itself as opposed to one of the Web server instances listed under the server node visible in the IIS MMC.

Configuring Site Member and Site Developer User Authentication

The method of authentication used for developers often needs to be different than the method used for Web site members. The most common scenario is one where members of a Web site authenticate themselves by entering their user ID and password into an HTML FORM in a Web page whereas developers use a Web authoring tool that is incompatible with HTML FORMs authentication. Developers commonly use HTTP Basic Authentication with WebDAV or WEC letting the HTTP client send user ID and password HTTP headers in each request. Development tools like Visual Interdev support HTTP Basic Authentication but not HTML Forms. For simplicity, it is usually okay to use the same database for managing user accounts as you use for managing developer accounts even if the method of authentication is different for the two, assuming you have a way to create groups of users and map those groups to Windows groups for compatibility with DACLs. It is also not uncommon to use built-in Windows user accounts for both members and developers, particularly in smaller Web user communities.

Microsoft Site Server's Personalization and Membership (P&M) feature gives you a user account system based on Lightweight Directory Access Protocol (LDAP) that is compatible with other server software such as Microsoft's SMTP and POP3 mail servers and is simple to integrate into any Web site hosted by IIS. It also supports HTTP Basic Authentication through an ISAPI filter DLL and is easy to use for HTML FORMs authentication with a little ADSI and ASP code to implement user login. However, there are many other ways to create the same user account features. If you don't want to use Site Server, you can still use its user account architecture and API for developing your own user account feature because Site Server P&M uses ADSI and LDAP to store and retrieve user account information. ADSI includes an LDAP provider for accessing LDAP servers. As long as you store user account data in an LDAP-compatible directory server, ADSI can be used within ASP script to create and authenticate user accounts without relying on P&M. The only piece you would miss is the ISAPI filter DLL provided by P&M for supporting HTTP Basic Authentication with LDAP-based user accounts. Without the P&M ISAPI filter for membership authentication, site developers have to have Windows user accounts.

The following instructions assume that you've decided to use Site Server P&M or have a replacement available for the P&M Membership Server which provides an underlying LDAP server. The process of binding to the LDAP server using ADSI to create a new user account should be the same as the example shown here. However, the instructions for automating the mapping of a Membership Server instance to the new Web server instance will only be applicable to a Site Server P&M installation, since those steps are specific to the configuration of the HTTP Basic Authentication ISAPI filter provided by P&M.

To create a new user in an LDAP directory service with ADSI, your code must first bind to the members container in the directory. Next, the ADSI core interface `IADsContainer::Create` method is called to create a new instance of the "Member" class in the container. A GUID is generated to uniquely identify the user internally within the LDAP directory service, and a password is assigned. Finally, the new user account is added to the group that will be used to grant Web site authoring permission: `AdminGroup`. The `AdminGroup` group corresponds to a Windows group created by the Site Server P&M Membership Server,

and is the same group that must be granted authoring permission when WebDAV or WEC are configured for the site.

```
Set objLDAP = GetObject("LDAP:")
SMembersDN = "LDAP://server/o=org/ou=Members"
sID = "LDAPAdminID"
sPW = "LDAPAdminPW"
sNewUserCN = "newusercn"
Set objMembers = objLDAP.OpenDSObject(sMembersDN,sID,sPW,0)
Set objMember = objMembers.Create("Member","cn=" & sNewUserCN)
Set objGuidGen = CreateObject("Membership.GuidGen.1")
objMember.Put "GUID", objGuidGen.GenerateGuid
objMember.userPassword = "password"
objMember.SetInfo
sGroupDN = "LDAP://server/o=org/ou=Groups/cn=AdminGroup"
Set adsGroup = objLDAP.OpenDSObject(sGroupDN,sID,sPW,0)
Set adsMemberOf = adsGroup.Create("memberof","cn=" & sNewUserCN)
sUserDN = "cn=" & sNewUserCN & ",ou=members,o=org"
adsMemberOf.Put "memberof",sUserDN
adsMemberOf.SetInfo
```

In the example shown here, the ID of the new user, or common name (CN) as it is known in an LDAP directory service, is newusercn. Adding this new user to the AdminGroup group is accomplished by creating a new instance of the memberof class inside the directory service's Groups container under the AdminGroup key. When you see LDAP://server/ in the code that address must point at a real LDAP server accessible locally or over the network or ADSI will be unable to contact the server. Your LDAP server will be configured to host a directory belonging to a particular organization namespace and that namespace must be part of the LDAP://server/ URL. The o=org is a placeholder in the sample shown for the organization namespace of your LDAP server. Finally, the variable sUserDN is set to the distinguished name (DN) of the user account, which is another term and naming convention specific to LDAP directory services.

Configuring Membership Authentication

Once you have developer user accounts in your membership directory, you need to configure your Web site to use the membership server for authentication. By default all new sites use Windows native user accounts for authentication. If you turn off Allow Anonymous on a directory or file published on a Web site, IIS will force the browser to supply authentication credentials before allowing access. Under Windows Authentication, a valid user ID and password of a local or domain user must be supplied and the user or a group the user belongs to must be granted the proper NTFS permissions otherwise IIS refuses the browser access to the password-protected file or folder. With Membership Authentication enabled instead, Site Server P&M's ISAPI filter for authentication takes over the task of authenticating credentials provided by the browser and a single Windows user account for impersonation is used for all requests made by authenticated users. Instead of IUSR_MachineName, which is the default impersonation account used for anonymous requests, another impersonation account specific to the membership directory is used by IIS for authenticated requests such as MemProxyUser1.

Each Web site that is configured for Membership Authentication can use a different membership server for verifying user credentials. This makes it possible for many user communities to exist within sites hosted by a single installation of IIS and Site Server. To enable Membership Authentication and select the P&M Membership Directory to use for the site, you can manually use MMC and right-click on the Web site instance, choose Tasks and then Membership Server Mapping and you'll see a list of the membership servers configured on the system. When Membership Server Authentication is configured for a Web site by setting the Membership Server Mapping via MMC, three changes are made to the Metabase for the Web site. First, a new DWord property with an ID number of 300 is added to the following key, where instanceid is the numeric identifier assigned to the Web site instance. Recall that an arbitrary instance ID of 100 was used in the earlier example script for creating a new Web site. Instance IDs must be unique but there is no special significance to the value of 100, it is actually used only as a unique textual identifier.

IIS://Localhost/Membership/Mappings/W3SVC/instanceid

Whereas a Web site's instance ID is arbitrary and of no special significance, the property ID of 300 is important to the way that software reads values from the Metabase. When IIS or the Site Server P&M Membership Authentication ISAPI filter need to determine which Membership Server instance a particular Web site is mapped to, it knows where in the Metabase to look because it constructs the same IIS:// path just shown, but that only references a key not a property in the Metabase. Properties are contained within keys and keys can be of certain KeyTypes. Each property type is assigned a unique identifier so that software that stores and reads those properties is able to reference them after opening a particular key. A programmer at Microsoft had to decide at some point to use 300 as the property ID for the DWord property that indicates the membership server instance to which a particular Web site is mapped.

However, VBScript isn't able to use ID number identifiers to reference Metabase properties using ADSI. VBScript is limited to using name identifiers like the ones used to reference KeyTypes. Therefore, if you wish to use ADSI from within VBScript to edit the DWord value of the membership server mapping Metabase property for a Web site, you need to take one extra step ahead of time to assign a name in the Metabase to any property identified by the ID number 300. You make this change to the Metabase schema using the Metabase Editor, MetaEdit.

Editing Metabase Classes and Properties

The Metabase Editor (MetaEdit) is a utility provided by Microsoft that enables you to manually read and edit the IIS Metabase. Knowledge base articles Q240225 and Q232068 describe MetaEdit and provide a link to download the latest version. MetaEdit is also supplied as a standard part of Windows Server 2003. When you open MetaEdit you see two top-level Metabase nodes, LM and Schema. LM is LocalMachine just as in the Registry. LocalMachine is a synonym for the "Localhost" that is commonly used in IIS:// URLs. Under LocalMachine are all of the settings for the IIS-hosted sites configured on the system. Under Schema are the Classes and Properties where the definitive list of Metabase key and property types is stored.

Expand the Properties node and right-click on Names. Choose New from the popup menu then select String. In the Edit Metabase Data window that appears, type 200 in the ID field and ServerName in the Data field then click OK. Add another String value with ID 300 and Data value ServerID. Figure 15-4 shows the resulting addition to the Metabase Properties by Name for ServerName and ServerID. Now you need to add a new Class so that you can create membership server mapping objects in the Metabase using VBScript and ADSI.

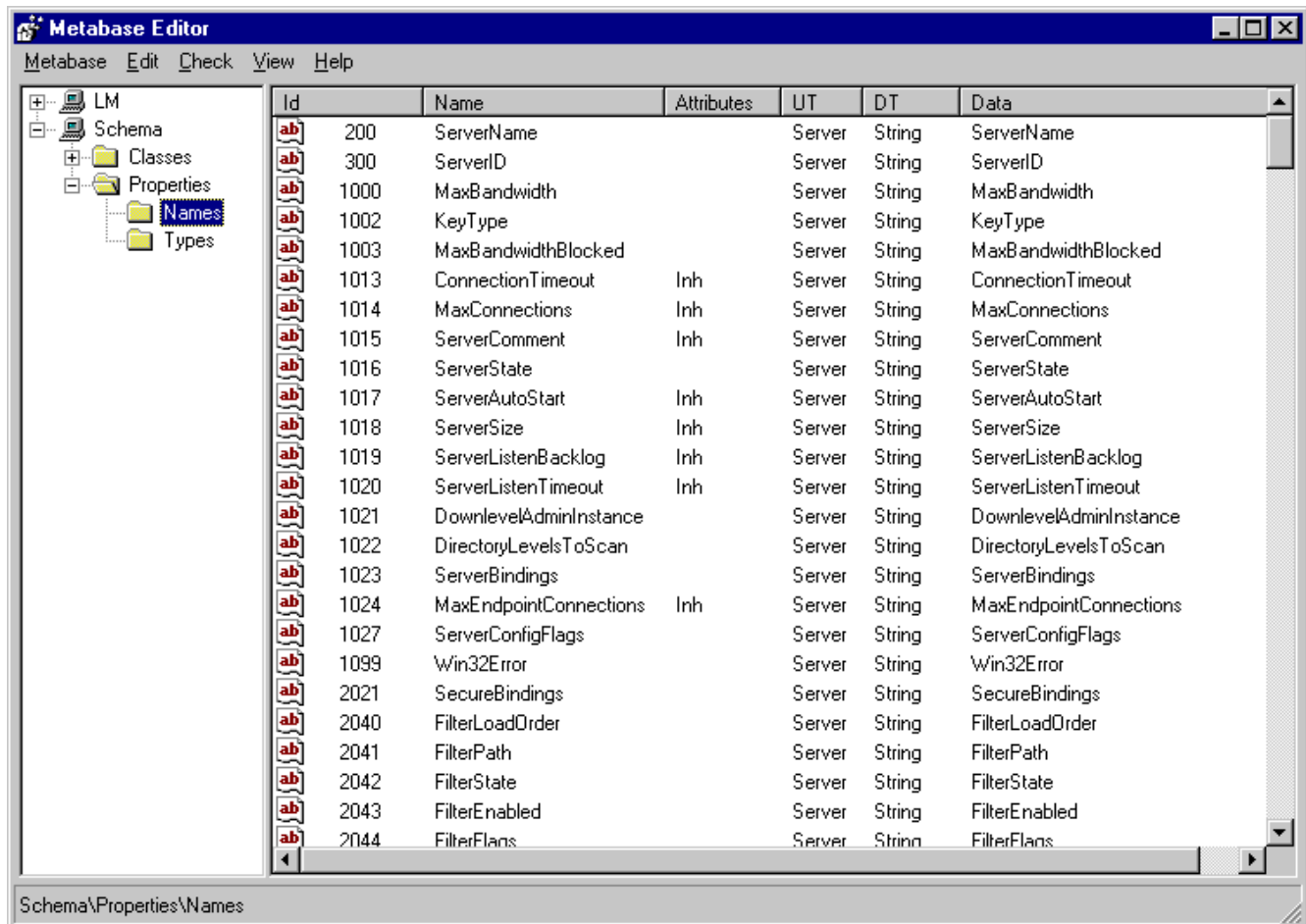


Figure 15-4: Use The Metabase Editor to Add ServerID and ServerName Property Names

Select the Classes node and right-click to access the popup menu. Choose New and then select Key to create a new key. Name the key IIsMembershipMapping and then right-click on the new key to access its popup menu. Choose New and then select Key again to create a subkey under the new key. Name the subkey "Mandatory". Then right-click on the key again and create another subkey named "Optional". The two subkeys should appear at the same level directly beneath the new IIsMembershipMapping key. Now add a String value to the Optional subkey and select ServerName from the ID drop-down list box. Leave the Data field blank. Since you previously added ServerName to the Properties of the Metabase, you can now use ServerName as a property type for new values you create in the Metabase. The ID drop-down contains a complete list of all known Properties including any custom ones that you define. Add a new DWORD value

to the "Optional" subkey but instead of selecting from the drop-down type in the ID number 300. Enter zero in the Data field or MetaEdit won't accept your new DWORD value since nulls aren't allowed in numeric properties. Figure 15-5 shows what your IIsMembershipMapping class should now look like in the Metabase.

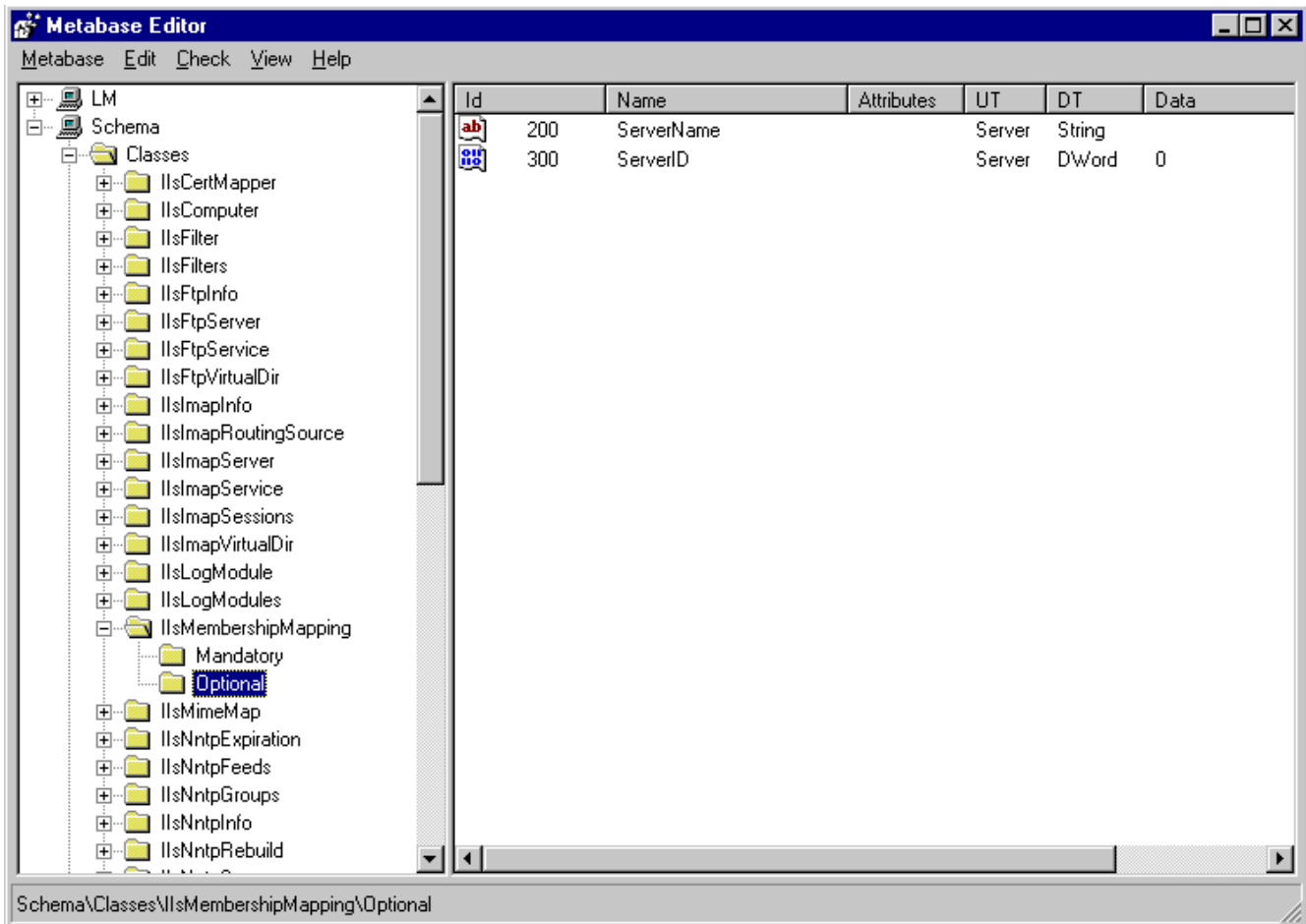


Figure 15-5: Use the Metabase Editor to Add an IIsMembershipMapping Class

With your new Metabase class configured, you're ready to automate Membership Server mapping configuration of a Web site. The following code is necessary to bind to the Membership Mappings Metabase key and create a new IIsMembershipMapping entry. The optional ServerName property can be left blank, it's for display only. But ServerID must be set to the instance ID of the Membership Server to which you want to map the Web site's authentication service. You will have to determine the appropriate instance ID manually or write code to discover the instance ID at runtime. Once a Membership Server is created its instance ID doesn't change so most applications will find it acceptable to manually determine the instance ID and hard-code it in Web site automated setup scripts.

```
Set mm = GetObject("IIS://Localhost/Membership/Mappings/W3SVC")
Set mmObj = mm.Create("IIsMembershipMapping", "100")
mmObj.ServerID = "13"
mmObj.SetInfo
Set wsObj = GetObject("IIS://Localhost/W3SVC/100")
wsObj.MembershipServerMapped = 1
```

```

wsObj.MembershipServerID = 13
wsObj.SetInfo
Set fs = wsObj.Create("IlsFilters","Filters")
fs.FilterLoadOrder = "Auth Filter"
fs.SetInfo
Set afObj = fs.Create("IlsFilter","Auth Filter")
afObj.FilterPath = "C:\Microsoft Site Server\bin\P&M\authfltr.dll"
afObj.FilterEnabled = 1
afObj.SetInfo

```

Shown also in the sample code are additional property values named MembershipServerMapped and MembershipServerID that must be set on the IIsWebServer object. These properties are created for you when Site Server is installed and you don't need to assign names to their respective property IDs in order to reference them from within VBScript. MembershipServerMapped is a Boolean flag that tells IIS that a Membership Server Mapping exists for the Web site. And, like the ServerID property you created in the previous step, MembershipServerID is used to indicate the instance ID of the mapped Membership Server. In MetaEdit you'll see that these two properties correspond to property ID numbers 2117 and 2200, respectively, as opposed to ID numbers 200 and 300 for ServerName and ServerID. Getting the right property ID into the right Metabase key and setting its value are the essential steps required to configure any aspect of IIS Metabase settings.

The final step to finish the Membership Mapping for a Web site is to install the ISAPI filter for Membership Authentication as shown in the sample code. An instance of the IIsFilter Metabase class is created under the Filters key for the Web site instance. The Filters key is an instance of the IIsFilters Metabase class. FilterLoadOrder tells IIS in what sequence to load ISAPI filters when more than one is present. The sample uses authfltr.dll from Site Server P&M. Even if you aren't using P&M, understanding how the Metabase is constructed based on its underlying schema with classes and properties is very important when you begin scripting Web site and publishing point provisioning with ADSI and the IIS Admin Objects.

FTP Service Publishing Point Provisioning with ADSI

Creating a new FTP site with ADSI is simply a matter of binding to the parent MSFTPSVC Metabase key, creating a new instance of IIS Admin Objects class IIsFtpServer and then creating an instance of IIsFtpVirtualDir within this new container object. The following sample shows how this works. The numeric identifier assigned to the new FTP publishing point in this sample is 13, and after creating its ROOT IIsFtpVirtualDir the Start method is called to launch the new IIsFtpServer.

```

Set FTPSVCObj = GetObject("IIS://localhost/MSFTPSVC")
Set PubPointObj = FTPSVCObj.Create("IIsFtpServer","13")
PubPointObj.ServerBindings = ".*:21:"
PubPointObj.AllowAnonymous = 0
PubPointObj.ServerComment = "New Publishing Point"
PubPointObj.ServerAutoStart = 1
PubPointObj.SetInfo

```

```

Set FTPd = PubPointObj.Create("IIsFtpVirtualDir","ROOT")
FTPd.AccessFlags = 1 ' Read only
FTPd.Path = "d:\inetpub\wwwroot"
FTPd.SetInfo
PubPointObj.Start

```

When creating a new FTP site with ADSI it's important to configure security settings such as AllowAnonymous (1=yes, 0=no) and carefully select ServerBindings values that expose the publishing point on the correct network interface and TCP/IP port number. The preferred DACLs for Web site content accessible via FTP publishing point similar to the one created in this sample (where Path is set to an \inetpub\wwwroot directory so that the FTP site shares its content with one or more Web sites) will depend on the users and groups present on your IIS box and the way in which you have assigned privileges. Aside from these security considerations with respect to FTP site configuration you may wish to configure DACLs on the Metabase key created under MSFTPSVC using MetaAcl as described previously, especially in a situation where multiple Web site administrators are given control over different parts of the Metabase. Figure 15-6 shows the Metabase keys and values created by the sample code. Note that values such as Win32Error and ServerState are added automatically by IIS when PubPointObj.Start is called to launch new IIsFtpServer 13.

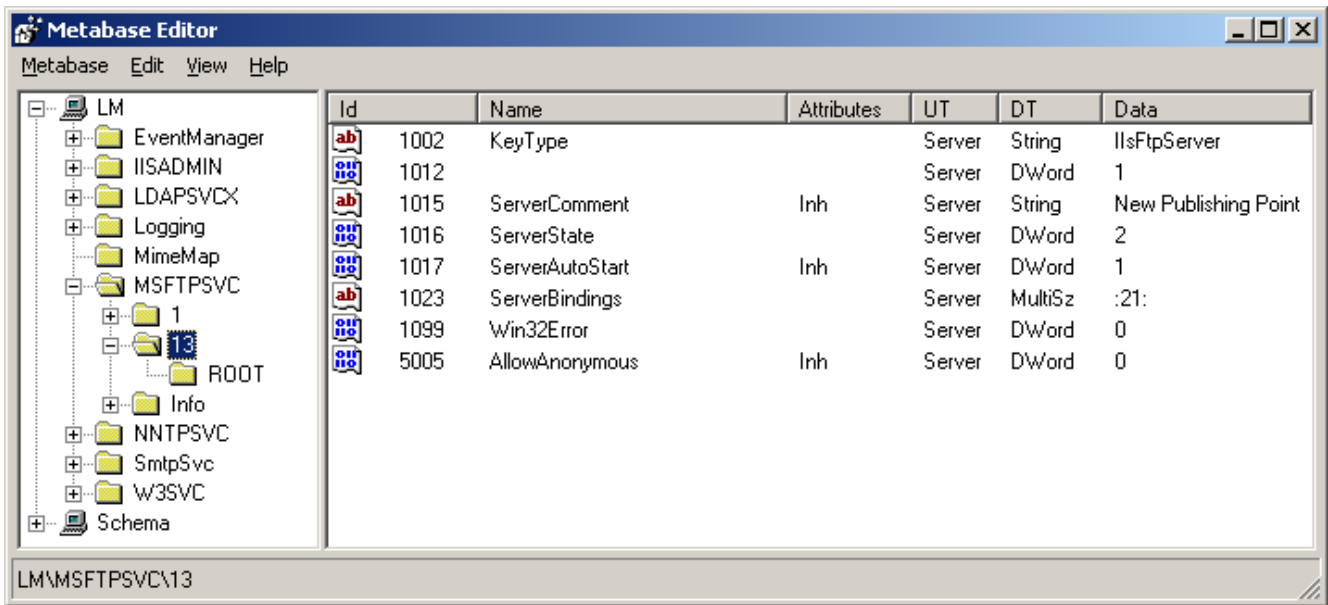


Figure 15-6: The Metabase View of MSFTPSVC IIsFtpServer 13

Creating or renaming FTP virtual directories beneath ROOT to implement a system of dynamic hidden virtual directories is easy based on the sample ADSI script. When IIS 6 is used you can also automate configuration of FTP user isolation mode. There is a new property of the IIsFtpServer class named UserIsolationMode as shown below. Because the IIS 6 Metabase is XML-based, it is somewhat easier to review the configuration settings and security properties of every publishing point.

```

<IIsFtpServer Location ="/LM/MSFTPSVC/13"
ServerAutoStart="TRUE"

```

```
ServerBindings=":21:"  
ServerComment="User Isolated FTP"  
UserIsolationMode="1">
```

The XML Metabase also makes ADSI scripting for automating management and security hardening far simpler because both the Metabase schema and the Metabase itself are much easier to understand and analyze. If you prefer to use programming languages other than scripting or tools that are not COM Automation-compliant so that the IIS Admin Objects are inaccessible to your code, the XML Metabase allows you to make configuration changes anyway through direct modification of these XML text files. The IIS Admin Objects and ADSI remain flexible and extensible under IIS 6 so that there is full bidirectional compatibility between Metabase changes made directly in XML and changes made with ADSI.

The standard tools, protocols, and services provided for use with IIS by managers of publishing points are lacking in many areas. IIS 6 FTP user isolation mode improves the useability of IIS FTP substantially, but the fact remains that FTP sends credentials over the network in cleartext form without encryption or hashing. Whenever FTP is used as a remote publishing point management interface it is essential that FTP user passwords be changed frequently. Ideally you would create a simple Web interface that issues time-limited credentials automatically to authorized users. Because the FTP protocol does not reauthenticate once an FTP session is established, the timeout on these credentials can be short enough to allow only a single use. When combined with dynamic FTP virtual directory paths and a two-stage publishing process where a different secure Web interface with a different set of credentials must be used to authorize publishing of content delivered to the server via FTP nearly every concern over eavesdropping and credential theft can be eliminated with minimal effort. This method of granting remote access to IIS publishing points may be preferred over allowing any of Microsoft's bloated remote publishing alternatives to put your IIS box at risk. WebDAV may be perfectly safe, or it may be as big a mess internally and architecturally as were the FrontPage Server Extensions. Only time will tell.

Chapter 16: Proving Baseline Security

What tools do you need to prove security? Software quality assurance testers can't even prove that they've conducted a comprehensive test of every execution path possible through the software whose quality they are charged with assuring. In most cases there are error handling conditions in program code that testers can't get at without staging some sort of failure simulation. And even then who knows for sure that the simulation matches the full severity and scope of real-world failure scenarios? How can all possible failure combinations be tested in advance, prior to shipping code to end-users? The answer is that information security quality assurance testing is nearly impossible using conventional software testing methodologies that focus on demonstrating the presence of features rather than proving security by demonstrating the absence of unintended functionality.

Execution of untested code is no different from execution of arbitrary malicious code. The only way we will ever know that our information systems are secure is to supplement our operating systems with a layer of protection that prevents the execution of code that has not previously been tested and proven to be safe. Bugs will still occur, of course, and bad data will always cause good programs to make what a human would consider to be a mistake. Security is not a mistake. To achieve it, prove to yourself you had it at some point in the past, or determine if you still have it, requires specialized tools used for infosec forensics.

We expect software developers to test their products before shipping them to us, but we don't require any proof they have done so. If they did provide us with such proof, it would be useless without tools that reproduce their testing methods, interpret the data, and enforce code execution restrictions based on the execution profile that quality assurance testing proved to be trustworthy. I would love to show you how to use these infosec forensics tools to analyze your binaries before deploying them in a production information system, but they do not yet exist. Sadly, we aren't even to the point, technically or procedurally, where we can prove that the code spinning electrons and magnetic polarities around in our microprocessors and persistent storage devices is code that our trusted vendors or our own programmers created. You might be inclined to conclude from this sad state of affairs that information systems are just too immature to be useful for important applications in a complex, high-risk environment like a large computer network or a network of networks like the Internet. You might be right, if somewhat conservative, in forming such an opinion. This does not change the fact that there's work to be done, and that it's a reasonable risk to rely on inherently untrustworthy computers to do that work. Awareness of security risks makes you recognize the need for better security tools and better infosec forensics procedures for proving and communicating low-level technical security details.

As customers we should not pay for products that are being tested on us. We should pay for products that have already been tested, and we should be given the results of that testing to use as a tool of security auditing and threat containment. A forensic profiling system for compiled code would enable numerous countermeasures to the threats that arise today out of the necessity to leave our microprocessors and OS APIs open to arbitrary

utilization. These resources can and should be closed to the run-time execution of code that does not have an accompanying forensic profile created by the developer as they carefully tested each logical path through the authentic compiled product. The only reason this is even a problem today is that everyone is looking at the complexity and unpredictability of computing from the programmer's perspective. Look at it from the perspective of a forensic analyst and you'll conclude that this complexity and unpredictability is only necessary during development and debugging. A programmer must be able to push arbitrary machine code instructions that have never been seen before into the microprocessor. Nobody else needs this ability, and nobody else benefits from it except attackers.

Untested execution paths through software are always error conditions. Software should gracefully self-destruct when it encounters such an error condition rather than assume it's okay to keep going. The OS should force software that refuses to gracefully self-destruct to do so when it starts to run outside of its pre-authorized static bounds. This isn't hard stuff to implement. It wouldn't require any change in hardware. The enabling technology for this stuff; the knowledge, ideas, research, and proof of concept algorithms to make it possible; revolve around computer forensics. Arbitrary malicious code can cause a CPU to do math, but it can't cause a CPU to do harm unless it is able to communicate with or control a willing victim such as a device driver. Tools used by forensic analysts are able to profile the expected execution paths of uncompromised computer programs. A properly-designed secure operating system could easily rely on forensic security templates to identify and control access to critical resources that might enable malicious code to do harm. Microsoft .NET provides something of a first step in this direction by introducing evidence-based computing and the ability to analyze run-time call stacks and make security decisions using real-time infosec forensics. But the first steps have barely even started that will eventually extend computer forensics to its logical conclusion.

The security tools we have available to us today to protect IIS are terrible. They're not just inadequate, they're flawed and vulnerable. This does not mean they are useless, nor that you should not use them. In fact, you must use them. Before doing so you should conduct your own forensic analysis of their design and function, and recognize the value they can offer while learning to manage the risks they create. It should not be the case that your IIS security tools themselves need additional security hardening, but then computers should not be programmable in the hands of users, programmers shouldn't make mistakes, and systems shouldn't be so complex that flaws are obscured by a tidal wave of features. A good rule of thumb is to assume that there is at least one serious security vulnerability for each unnecessary feature. More vulnerabilities occur in software that is designed on purpose to allow different parts written by different programmers to overlap. Whenever possible, build your own security analysis tools using knowledge and information that you can easily prove to be reliable or discover to be flawed. Use other people's code only to enhance and extend the variety of analysis you are capable of performing. And start with a known-good baseline of up-to-date authentic code from a trustworthy source.

Microsoft Baseline Security Analyzer

Keeping up-to-date on hotfixes and service packs is considered one of the pillars of information security. Equally important is keeping up-to-date on vulnerability

announcements and technical details behind them which lead to the necessity for bug fixes.

If you don't read every word published by Microsoft's security group and analyze every security-related knowledge base article then you have no hope of understanding the threats that exist in the real world and the things that are being done to counter those threats. Without this understanding you may as well let somebody else manage the security of your IIS boxes. Your most important security tool is the browser that enables in-depth review of knowledge base articles and keeps you plugged-in to the infosec community and Microsoft's role in this community. Because security bulletins and knowledge base articles can be issued or revised without your knowledge, a security coordinator who provides focused, edited security information and a comprehensive list of security-related knowledge base articles and bulletins is a critical part of your ongoing security process. Shavlik Technologies (shavlik.com) developed the Microsoft Baseline Security Analyzer (MBSA) to provide all Windows users with the benefits of a human security coordinator while customizing and automating the security analysis steps recommended by Microsoft.

MBSA can be viewed as the centerpiece of your routine security assurance procedures, but it's important to remember that MBSA is most likely vulnerable to various exploits just like any other software. Better than letting MBSA execute while you sit there with a blank stare hoping for the best is to carry out as much of the analysis yourself that MBSA would normally perform for you. When you do run MBSA, there are things you should do to minimize risk associated with this program. Figure 16-1 shows the welcome screen that appears when you install and run the program. Since well-known security tools are obviously prime targets of attack and tampering, it's useful to review the internal architecture of any such tool that you use.

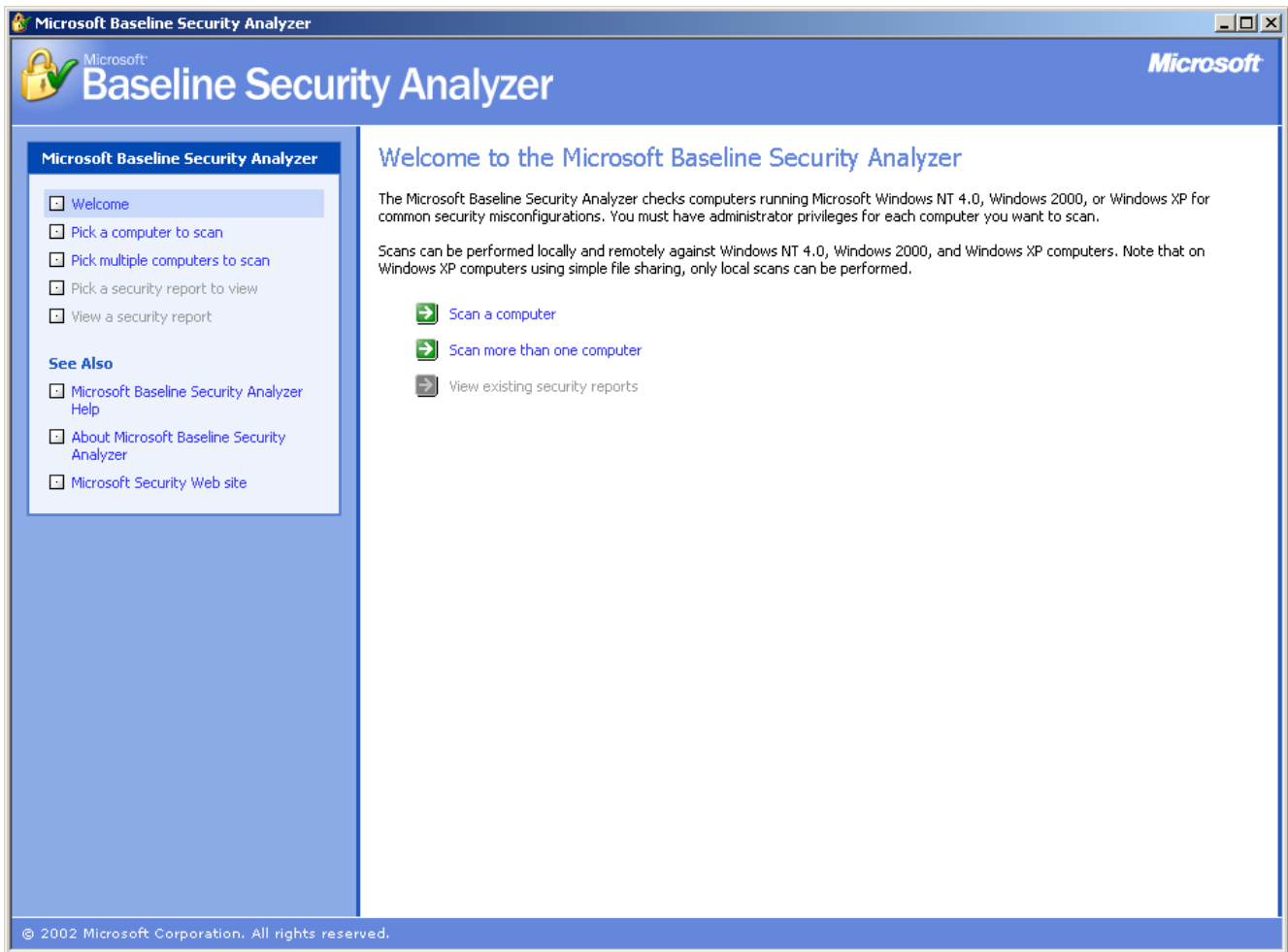


Figure 16-1: Microsoft Baseline Security Analyzer

The user interface of MBSA is built using HTML, which represents a threat in and of itself. Because the MBSA code attaches itself to the Web browser that is used to view the MBSA HTML components, vulnerabilities in your browser platform can be exploited easily through modifications to the MBSA HTML files. Each UI screen is a separate file stored in the directory that was selected when MBSA was installed. The UI shown in Figure 16-1 is comprised internally of `Welcome.html`, the `Scanner.css` stylesheet, `ToolBar.html`, `Header.html`, `Footer.html`, and `Default.html`. This HTML UI application design greatly simplifies internationalization and has other practical benefits, but it creates additional uncertainty with respect to security. One of the most important features of MBSA is its ability to verify that the software installed on a box is the most up-to-date authentic code published by Microsoft. But MBSA makes no attempt to verify the authenticity and up-to-date status of HTML data. Because few local applications rely on HTML for UI and application functionality, the concept that data files are now a crucial part of certain programs' code has yet to be incorporated into security tools like MBSA. As a result, it's necessary to take extra precautions before executing MBSA (`mbsa.exe`) and to revisit these precautions regularly.

There is no mechanism in place to protect the MBSA HTML files from tampering because HTML files are still not considered executable content under Windows in spite of the obvious threat they pose, particularly when hosted and interpreted in the local intranet

zone as part of a stand-alone program like MBSA. When you scan a computer with MBSA, client side javascript calls methods on an object exposed by ServerSecure.dll that gets created by the following <object> tag:

```
<object ID="Scanner" CLASSID="clsid:46CDA2F0-24A4-4b8d-991A-7250C376FD44">
```

The first method call is StartScan(parent.gblScanMode,parent.gblScanParams, parent.gblReportName) which is triggered by loading the page scan.html where the scanner object previously instantiated in the parent frame is configured for use based on the values of global variables. MBSA enables the scanning of a single or multiple computers. The pickcomputer.html page shown in Figure 16-2 is a single computer counterpart to pickcomputers.html and both files make use of MSXML2.DOMDocument through the following code:

```
try{  
var xml = new ActiveXObject("MSXML2.DOMDocument.4.0");  
var objxml = new ActiveXObject("MSXML2.DOMDocument.4.0");  
} catch (e) {  
var xml = new ActiveXObject("MSXML2.DOMDocument")  
var objxml = new ActiveXObject("MSXML2.DOMDocument") }
```

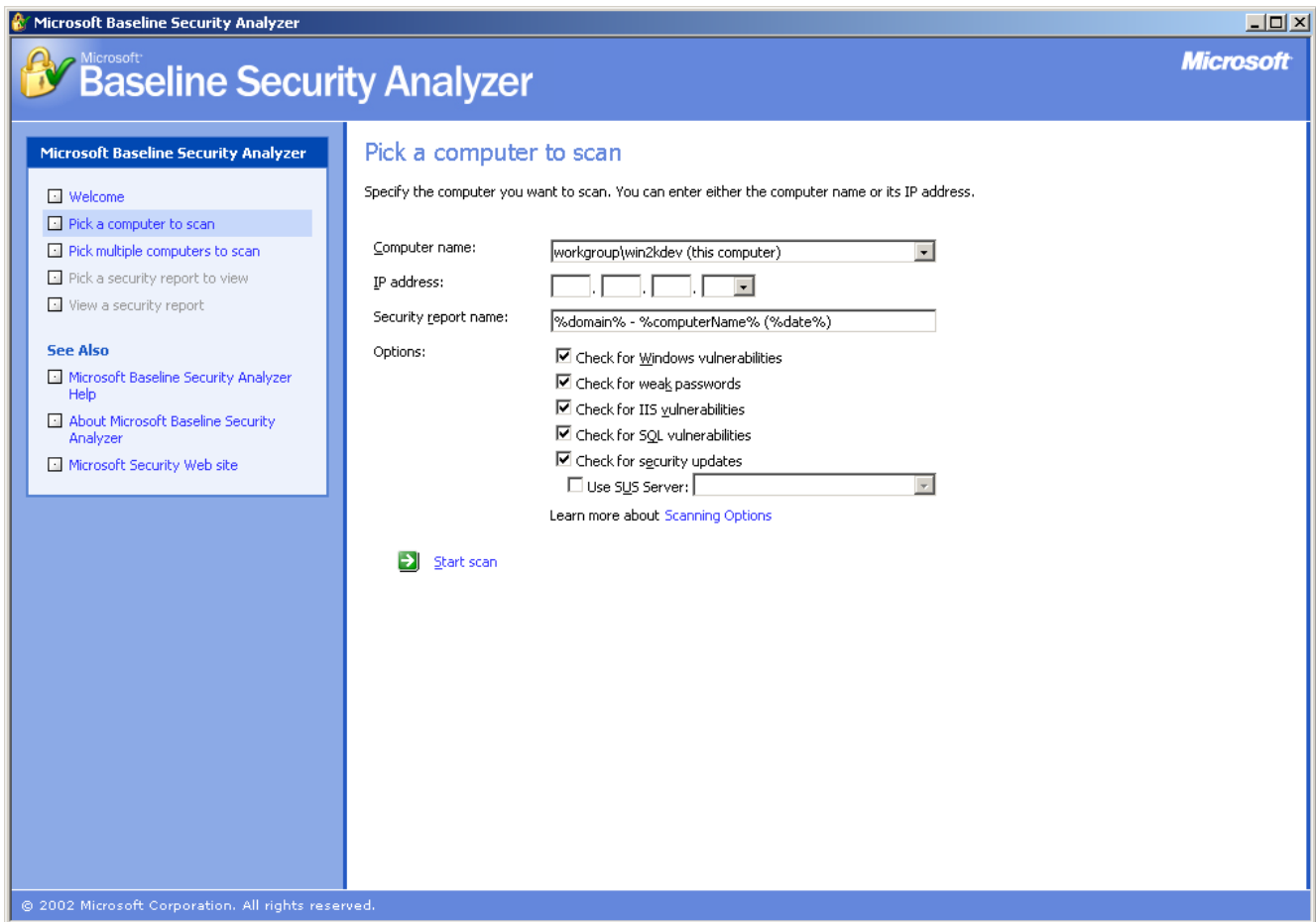


Figure 16-2: Scanning a Computer with MBSA Relies on HTML, Script, and ActiveX

By placing a check mark in the Use SUS Server check box, you can instruct MBSA to rely on a Software Update Services server. An SUS server maintains a list of authorized hotfixes and service packs that have been approved for distribution to Windows boxes within the enterprise. Rather than acting as a staging server and distribution point for authorized updates, and without adding the obvious security protection that should be present whereby authorized code distributions are digitally signed using your own private key so that the boxes under the control of the SUS server don't have to automatically trust any third-party signatures or root CA certificates, SUS functions simply by filtering the list of existing hotfixes. Supposedly all persons in control of Windows boxes in your organization will comply with the hotfix filter policy published by the SUS server. This brings up the obvious policy question: "Who gets fired when a critical hotfix is applied without permission from SUS and the hotfix prevents a devastating attack, the person who applied the unauthorized hotfix or the person who filtered it out using SUS?" If you choose to use SUS, beware of its inherent limitations. SUS addresses only the management and control complications caused when Windows Update is used to automatically install downloaded code from external Windows Update servers. SUS does not allow you to run your own Windows Update server, or replace Microsoft's digital signature with your own for removing inappropriate external trust dependencies, it simply gives veto power over automatic updates installed by Windows boxes that agree to be vetoed.

MBSA produces a security report for the specified local or remote computer (or computers) based on the scanning options selected. As long as you select "Check for IIS vulnerabilities" you will see a section in the report devoted specifically to security analysis of IIS. Figure 16-3 shows the Security Update Scan Results summary, an overall risk assessment, and the number of hotfixes and service packs that are found to be missing for each of the scanned subsystems. This security report is automatically saved and can be viewed at a later time by clicking on Pick a security report to view in the MBSA navigation menu. The IIS Security Updates portion of the report shown lists 1 critical security update as missing.

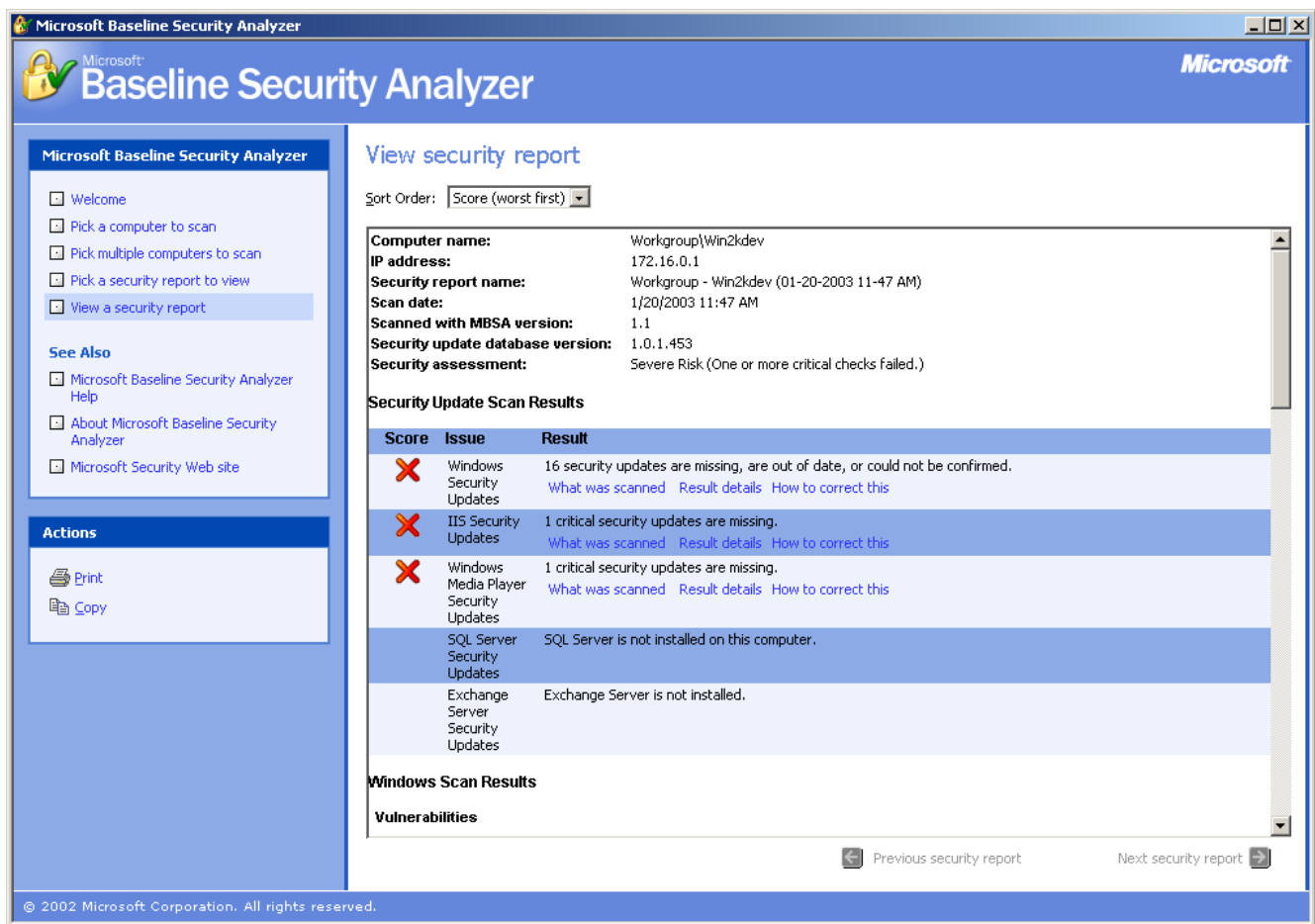


Figure 16-3: MBSA Produces Detailed Hotfix and Service Pack Scan Results

One of the reasons that MBSA is valuable, in spite of its various flaws, is that it automates the detection of well-known vulnerabilities and common configuration mistakes that occur in the wild. When MBSA reports that a vulnerability is present on your box, there is little doubt that it is right. On the other hand, you cannot be certain that a vulnerability is absent just because MBSA doesn't detect and report it. This is the most crucial point to understand about the design of MBSA and most any security analysis tool for that matter: only positive results are meaningful. Negative results do not prove anything, they just offer circumstantial evidence of a probable condition.

Well-Known Vulnerability Scanning with MBSA

Each vulnerability analyzed by MBSA receives a score. Either a green check mark (low risk, high probability of absent vulnerability) a yellow X (moderate risk, some minor vulnerabilities detected) or a red X (extreme risk, critical vulnerabilities detected). Informational items are assigned a blue asterisk score, indicating that risk is undetermined or not applicable to the report item. Figure 16-4 shows the result of MBSA scanning for well-known Windows vulnerabilities. Password, filesystem, and account restriction policies are analyzed among other sources of vulnerabilities that may exist even when a Windows box is running the most current hotfix or service pack version of each authentic Windows binary published by Microsoft.

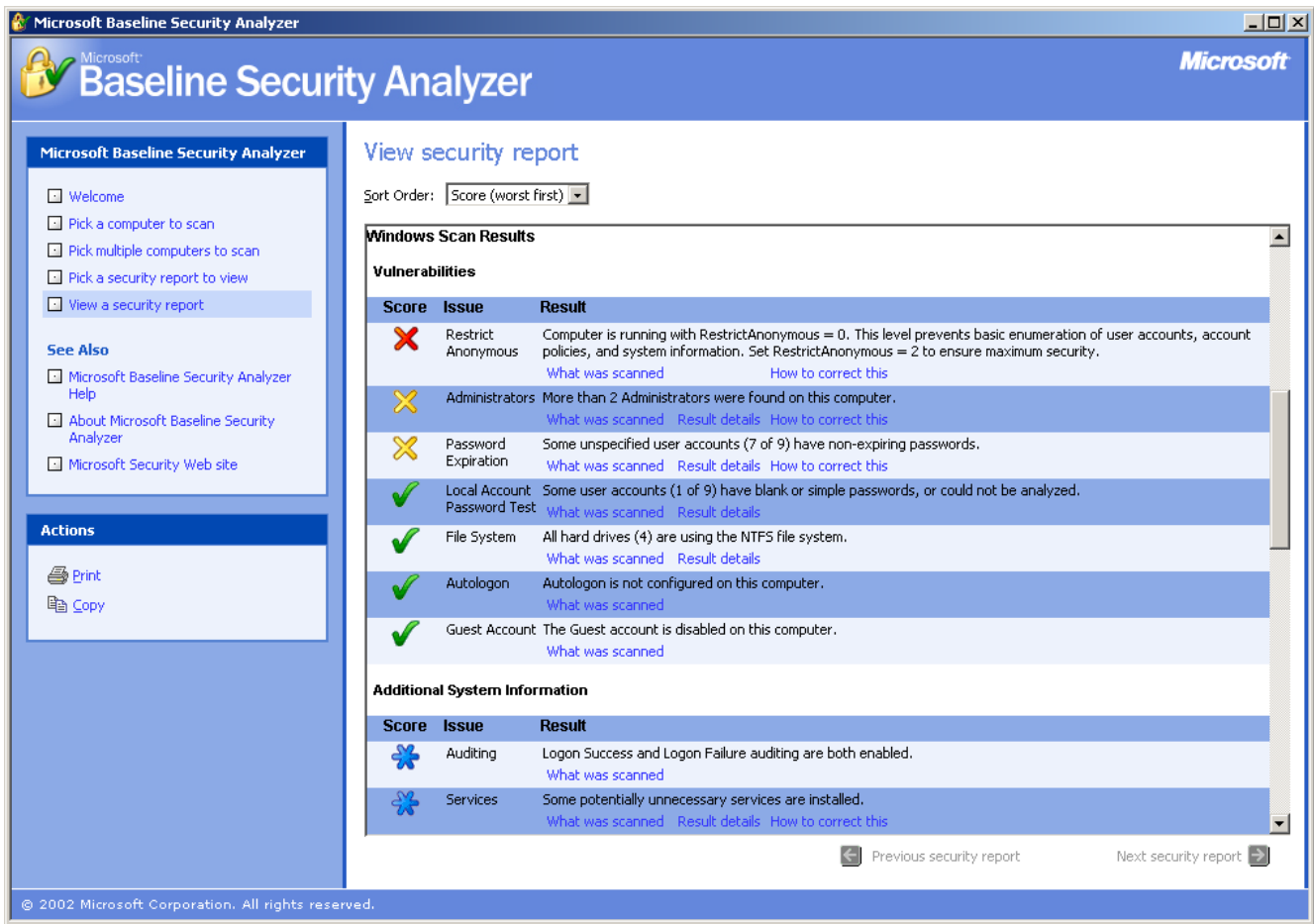


Figure 16-4: MBSA is Also Designed for Common Vulnerability Scanning

Vulnerabilities known to exist in IIS deployments can also be scanned with MBSA. The presence of sample Web applications, default virtual directories, and other risky configurations often found when IIS has not been properly locked down and hardened are flagged for further investigation. Figure 16-5 shows a typical report that suggests the presence of risky configuration settings in IIS and recommends that the IIS Lockdown Tool be run to further harden IIS against attack. Informational items flagged with an asterisk confirm that IIS is not installed on a domain controller, and supply detailed recommendations for optimum security logging settings.



Figure 16-5: Potential IIS Vulnerabilities are Analyzed Explicitly by MBSA

For each item in an MBSA report there are hyperlinks to additional pages containing details about what was scanned, why the results appear as they do, and what course of action to take if you wish to correct the detected vulnerability. This information is derived from best practices documents, knowledge obtained through real-world support services provided to customers that end up documented in the Microsoft Knowledge Base, suggestions found in security bulletins, and Microsoft's understanding of the internal technical architecture of the software products you use. New information and knowledge are constantly being released that make the security vulnerability scanning logic embedded in MBSA out-of-date. For this reason it is important to update MBSA as soon as possible when new releases become available. MBSA's analysis of the hotfix and service pack status of a Windows box is not part of its hard-coded scanning logic. Rather, MBSA is designed to download details of the latest hotfixes and service packs or read these details from a local file. In this way MBSA incorporates the functionality previously released as HFNETCHK, the network enabled hotfix scanner used to determine patch status from the most up-to-date list of hotfixes and security bulletins published by Microsoft. It is very important to understand where this information comes from and what it consists of, because it forms the technical foundation of baseline security scanning with MBSA. This foundation is in fact a human-edited subset of all hotfixes that Microsoft releases. There are procedural flaws in the way that this information gets assembled, and technical flaws in the way that it gets distributed to your box and used by MBSA.

Analyzing The XML Source Data for Baseline Security

Baseline security is established in MBSA through an XML file named mssecure.xml. This file is published by Microsoft in two different ways. The preferred publishing point is a digitally-signed cabinet file (mssecure.cab) that contains the compressed XML file. The mssecure.cab file is downloaded by MBSA using unencrypted HTTP and the download is therefore vulnerable to DNS hijacking or spoofing attacks and MITM exploits that will cause data of an attacker's choice to be downloaded instead of the latest digitally-signed .CAB file. To mitigate the threat this sort of attack might pose, Shavlik hard-coded into MBSA the requirement that the .CAB file contain a verifiable digital signature applied with either a Microsoft or a Shavlik Technologies code signing certificate. There may also be buffer overflow vulnerabilities or other bugs in this download-and-verify process that would give an attacker the ability to execute arbitrary code or force an arbitrary XML file to be trusted by MBSA in place of the authentic digitally-signed mssecure.cab. A known vulnerability in the design of MBSA gives a MITM the ability to rollback your current mssecure.cab by replacing it with a digitally-signed copy of an older version simply by causing your Windows box to retrieve the old version from a network node under the attacker's control.

The mssecure.cab file published by Shavlik Technologies is at this URL:
<http://xml.shavlik.com/mssecure.cab>

To get mssecure.xml from Shavlik Technologies (not the .cab file) use this URL:
<https://xml.shavlik.com/mssecure.xml>

Microsoft publishes a different version of mssecure.cab at the following URL:
<http://download.microsoft.com/download/xml/security/1.0/nt5/en-us/mssecure.cab>

To get mssecure.xml from Microsoft (not the .cab file) use the following URL:
<https://www.microsoft.com/technet/security/search/mssecure.xml>

Microsoft's international customer support groups publish their own language localized version of mssecure.cab and the Japanese version can be found at URL:
<http://download.microsoft.com/download/xml/security/1.0/NT5/JA/mssecure.cab>

When mssecure.cab can't be retrieved over the network at runtime, MBSA attempts to download mssecure.xml from an SSL-secured Microsoft Web site. There is no MITM protection enabled for the download of mssecure.cab, since SSL is not used, and the SSL server authentication employed for downloading mssecure.xml when mssecure.cab can't be retrieved relies on open-ended automatic certificate chain trust just like SSL in Internet Explorer. Instead of allowing you to specify a known good, trustworthy public key for authenticating the Microsoft Web server, as you would expect for maximum security, any certificate chain that offers an SSL certificate that appears valid will be accepted as authentic. This means that a private key compromise of any trusted root CA or a software bug that results in an attacker who is able to issue their own arbitrary SSL certificate enables successful MITM attacks against MBSA when it retrieves mssecure.xml via HTTPS. Protection offered by the digital signature applied to mssecure.cab is supposed to eliminate the need for SSL server authentication and encryption but clearly it does not. By design, MBSA accepts any version of mssecure.cab regardless of how out-of-date it

is. The only way to know that your mssecure.cab is the most current version is to download and review the file yourself before each use of MBSA. Do not let MBSA retrieve and verify either mssecure.cab or mssecure.xml automatically. Each MBSA analysis report contains the version number of the mssecure.xml file used to produce the report. When MBSA detects a newer version on the local filesystem, it flags all old reports that you view using MBSA with a note like the following:

Security update database version: 1.0.1.453 ** Newer version 1.0.1.456 is available **

The current version of mssecure.xml always differs between Shavlik Technologies' version and Microsoft's. In fact, the Microsoft and the Shavlik versions never synchronize because they are maintained by different groups of people who don't merge details of work done by the other. Also, various international customer support groups at Microsoft create language localized mssecure.xml files for users of international versions of Windows. With all these conflicting versions of mssecure.xml floating around, there are numerous ways for an attacker to gather and misuse mssecure.xml and corresponding digitally-signed mssecure.cab files. While you would expect MBSA to display a prominent warning when you attempt to scan your IIS box with an outdated version of mssecure.xml, no such warning exists. The only time you will ever see the ** Newer version is available ** message is when a newer version of mssecure.xml is present in the MBSA install directory and you view old reports produced in the past using MBSA. The database version number displayed in reports is found in a <BulletinDatastore> tag like the one shown below.

```
<BulletinDatastore      DataVersion="1.0.1.456"      LastDataUpdate="1/23/2003"
  SchemaVersion="1.0.0.11"      LastSchemaUpdate="6/6/2001"      ToolVersion="3.32"
  MBSAToolVer="1.0"      MBSAToolURL="http://www.microsoft.com/
  technet/security/tools/Tools/MBSAhome.asp"      RevisionHistory=".298 includes various
  sqnumber updates">
```

The <BulletinDatastore> tag shown has DataVersion number 1.0.1.456 and this is the version number that will appear in each MBSA report produced using this XML data. Version 1.0.1.456 of Microsoft's mssecure.xml file was published on 1/23/2003 for use with MBSA version 1.0. Many improvements are underway to MBSA, and Shavlik Technologies is always the source of the latest software build. You can see from the version of mssecure.xml published by Shavlik Technologies the day before (see the following XML) that Shavlik had already progressed to MBSA version 1.1 as of 1/22/2003 and that they follow a different DataVersion numbering scheme from the one that Microsoft uses. Because MBSA displays only a version number of the XML data file relied upon for baselin security scanning, if Shavlik and Microsoft ever publish mssecure.xml files with identical DataVersion numbers, even more confusion and potential for abuse would emerge.

```
<BulletinDatastore      DataVersion="1.1.1.589"      LastDataUpdate="1/22/2003"
  SchemaVersion="1.0.0.11"      LastSchemaUpdate="6/6/2001"      ToolVersion="3.86"
  MBSAToolVer="1.1"      MBSAToolURL="http://www.microsoft.com/
  technet/security/tools/Tools/MBSAhome.asp"      RevisionHistory="Shavlik MSSecure XML
  File">
```

Figure 16-6 shows the type of informative technical explanations that are shipped with MBSA to help you understand what it is designed to do and where its capabilities come from. You must read this documentation and question its veracity because for some reason Microsoft programmers prefer to make pretty user interfaces that give you a nice warm fuzzy feeling of security rather than force you to consider the flaws and limitations of the software they create. Without reading every page of documentation provided with MBSA, and clicking through to the details of each report item displayed, you can't trust what you see in MBSA reports because subtle details like the fact that many security bulletins never find their way into XML file mssecure.xml and therefore are ignored by MBSA just don't get acknowledged by the MBSA GUI. If you read between the lines in the detailed documentation pages, you can see clearly that MBSA is designed to be user friendly first. You will receive no warning other than this documentation as to the fact that MBSA, by design, stores a copy of mssecure.xml on the local hard drive and when a new copy cannot be retrieved from the Microsoft Web server MBSA will silently use the cached file. The only way that you will know that MBSA has failed to retrieve and use the most current version of mssecure.xml is to manually retrieve this file yourself and examine its <BulletinDatastore DataVersion> tag and parameter. This allows you to compare the version number of the most current mssecure.xml file with the version displayed in reports. You might manually place a copy of the most current mssecure.xml file in the MBSA subdirectory, but unless you invoke MBSA in command-line mode (using MBSACLI.EXE) instead of the GUI version, MBSA may overwrite the mssecure.xml file you place in its subdirectory with whatever version arrives over the network.



Figure 16-6: MBSA Obtains a List of Service Packs and Security Updates from mssecure.xml

The GUI version of MBSA also ignores checksum validation, scans only for baseline security updates, and suppresses security update check notes and warnings. Why would Microsoft ship MBSA in its weakest possible default configuration? The only explanation is that the MBSA tool was designed to give the user a warm fuzzy feeling of security, it was not designed to perform a sincere and thorough examination of each Windows box. You can see from the usage instructions shown below that the Microsoft Baseline Security Analyzer Command Line Interface (MBSACLI.EXE) makes no secret of its superior capabilities as compared to the default MBSA GUI. It is shameful that Microsoft

chose to distribute the GUI version of MBSA as a political tool of deception rather than as a visual front-end to all MBSACLI.EXE functionality.

MBSACLI.EXE Usage Instructions

Microsoft Baseline Security Analyzer

Version 1.1.0.5

(c) 2002, Microsoft Corporation. All rights reserved.

Developed for Microsoft Corporation by Shavlik Technologies, LLC

www.shavlik.com

MBSACLI [/c|i|r|d target] [/n option] [/o file] [/f file] [/qp] [/qe] [/qr]

MBSACLI [/e] [/l] [/ls] [/lr file] [/ld file] [/hf] [/?]

Description:

This is a command line interface for Microsoft Baseline Security Analyzer

Parameter List:

/c domain\computer Scan named computer.
/i IP Scan named IP address.
/r IP-IP Scan named IP addresses range.
/d domain Scan named domain.
/n option Select which scans to NOT perform.
All checks are performed by default.
Valid values:
"OS", "SQL", "IIS", "Updates", "Password".
Can be concatenated with "+" (no spaces).
/o filename Output XML file name template.
Default: %domain% - %computername% (%date%).
/f filename Redirect output to a file.
/qp Don't display scan progress.
/qe Don't display error list.
/qr Don't display report list.
/s 0 Don't suppress security update check notes and warnings.
/s 1 Suppress security update check notes.
/s 2 Suppress security update check notes and warnings.
/baseline Check only for baseline security updates.
/nosum Security update checks will not test file checksums.
/sus SUSserver Check only security updates approved at the specified SUS server.
SUS implies /nosum; include /sum after the /sus option to override
/e List errors from the latest scan.
/l List all reports available.
/ls List reports from the latest scan.
/lr filename Display overview report.
/ld filename Display detailed report.
/v Display security update reason codes.
/hf Hotfix Checker Run in HFNETCHK mode.
Run with /hf -? for HFNETCHK help
The /hf must be the first command line parameter.
/? Display this help/usage.

Executing MBSACLI with no parameters scans the local computer.

Examples:

MBSACLI

```
MBSACLI /n Password
MBSACLI /c MyDomain\MyComputer /n Password+Updates+SQL
MBSACLI /d MyDomain
MBSACLI /i 200.0.0.1
MBSACLI /r "200.0.0.1-200.0.0.50"
MBSACLI /l
MBSACLI /ld "Domain - Computer (03-01-2002 12-00 AM)"
MBSACLI /f "C:\results.txt"
MBSACLI /sus "http://corp_sus"
MBSACLI /hf -?
```

The MBSA V1.1 graphical interface default scan parameters are: /s 2 /nosum /baseline

The importance of supplementing MBSA with manual review of the XML file mssecure.xml is underscored by the circumstances of the outbreak of the Sapphire, or W32.SQLSlammer, worm on January 25, 2003. A previously-published Microsoft security bulletin, MS02-039, warned of the SQL Server buffer overflow exploited by this worm in July, 2002 and a fix for this vulnerability was released as Knowledge Base Article and hotfix Q323875. Shavlik Technologies updated mssecure.xml on xml.shavlik.com to scan for the latest version of the vulnerable file, ssnetlib.dll, while Microsoft did not do so until just days prior to the worm. Six months elapsed between the publication of hotfix Q323875 and its inclusion by Microsoft in the default version of mssecure.xml used by MBSA and HFNETCHK v3.32. SQL Server Service Pack 3 was also omitted from mssecure.xml as of January 23rd, making patch status assurance for Windows boxes running SQL Server more difficult than necessary, and further emphasizing the warm fuzzy feeling of security that MBSA was designed to provide to its users.

The Shavlik Technologies version of mssecure.xml contained proper information about SQL Server Service Pack 3 prior to the outbreak of Sapphire, while Microsoft's mssecure.xml did not at any time prior to the worm. Reviewing mssecure.xml and using the most up-to-date version available makes MBSA more valuable. By design, MBSA only knows of file updates detailed in mssecure.xml and by design you will be warned of only a subset of all known threats. Look at the details of missing security updates as shown in Figure 16-7 and you can see plainly that it is Microsoft security bulletins that imbue the power of observation to MBSA, and that you really don't get any security without reading these bulletins.

Microsoft Baseline Security Analyzer - Microsoft Internet Explorer

Baseline Security Analyzer

16 security updates are missing, are out of date, or could not be confirmed.

Result Details

Windows Security Updates

Security updates confirmed as missing are marked with a red X

Score	Security Update	Description	Reason
X	MS02-042	Flaw in Network Connection Manager Could Enable Privilege Elevation (Q326886)	File D:\WINNT\system32\netman.dll has a file version [5.0.2195.5282] that is less than what is expected [5.0.2195.5974].
X	MS02-045	Unchecked Buffer in Network Share Provider can lead to Denial of Service (Q326830)	File D:\WINNT\system32\xactsrv.dll has a file version [5.0.2134.1] that is less than what is expected [5.0.2195.5971].
X	MS02-048	Flaw in Certificate Enrollment Control Could Allow Deletion of Digital Certificates (Q323172)	The registry key **SOFTWARE\Microsoft\Internet Explorer\ActiveX Compatibility\{43F8F289-7A20-11D0-8F06-00C04FC295E1}** does not exist. It is required for this patch to be considered installed.
X	MS02-051	Cryptographic Flaw in RDP Protocol can Lead to Information Disclosure (Q324380)	File D:\WINNT\system32\drivers\rdpwd.sys has a file version [5.0.2195.5243] that is less than what is expected [5.0.2195.5880].
X	MS02-055	Unchecked Buffer in Windows Help Facility Could Enable Code Execution (Q323255)	File D:\WINNT\hh.exe has a file version [4.74.8793.0] that is less than what is expected [5.2.3644.0].
X	MS02-063	Unchecked Buffer in PPTP Implementation Could Enable Denial of Service Attacks (Q329834)	File D:\WINNT\system32\drivers\raspppt.sys has a file version [5.0.2195.4080] that is less than what is expected [5.0.2195.6076].
X	MS02-068	Cumulative Patch for Internet Explorer (324929)	File D:\WINNT\system32\wininet.dll has a file version [6.0.2715.400] that is less than what is expected [6.0.2718.400].
X	MS02-069	Flaw in Microsoft VM Could Enable System Compromise (810030)	File D:\WINNT\system32\msjava.dll has a file version [5.0.3805.0] that is less than what is expected [5.0.3809.0].
X	MS02-070	Flaw in SMB Signing Could Enable Group Policy to be Modified (309376)	File D:\WINNT\system32\localspl.dll has a file version [5.0.2195.5423] that is less than what is expected [5.0.2195.6090].
X	MS02-071	Flaw in Windows WM_TIMER Message Handling Could Enable Privilege Elevation (328310)	File D:\WINNT\system32\msgina.dll has a file version [5.0.2195.4733] that is less than what is expected [5.0.2195.6090].

Figure 16-7: Mssecure.xml is Constructed Manually from MS Security Bulletins and Q Articles

IIS security updates that are detected as missing by MBSA always correspond to a particular security bulletin. Figure 16-8 shows how MBSA provides details of missing updates specifically for IIS, and simplify access to the security bulletins themselves. This is the real strength of MBSA, its ability to function as a specialized security bulletin browser. Is there really a need for another Web browser interface to the Microsoft security bulletins in the first place? Does it make sense to publish an outdated subset of bulletins assembled as an XML file by a human editor when the most current list of bulletins is always available on the Microsoft security Web site anyway? Wouldn't a tool that reads a comprehensive list of authentic hash codes for all known good Microsoft binaries and uses this list as the basis of comparison against hash codes computed for each file present on a Windows box be a far better tool for ensuring security and patch status of Windows boxes? These questions were apparently never asked by the developers of MBSA.

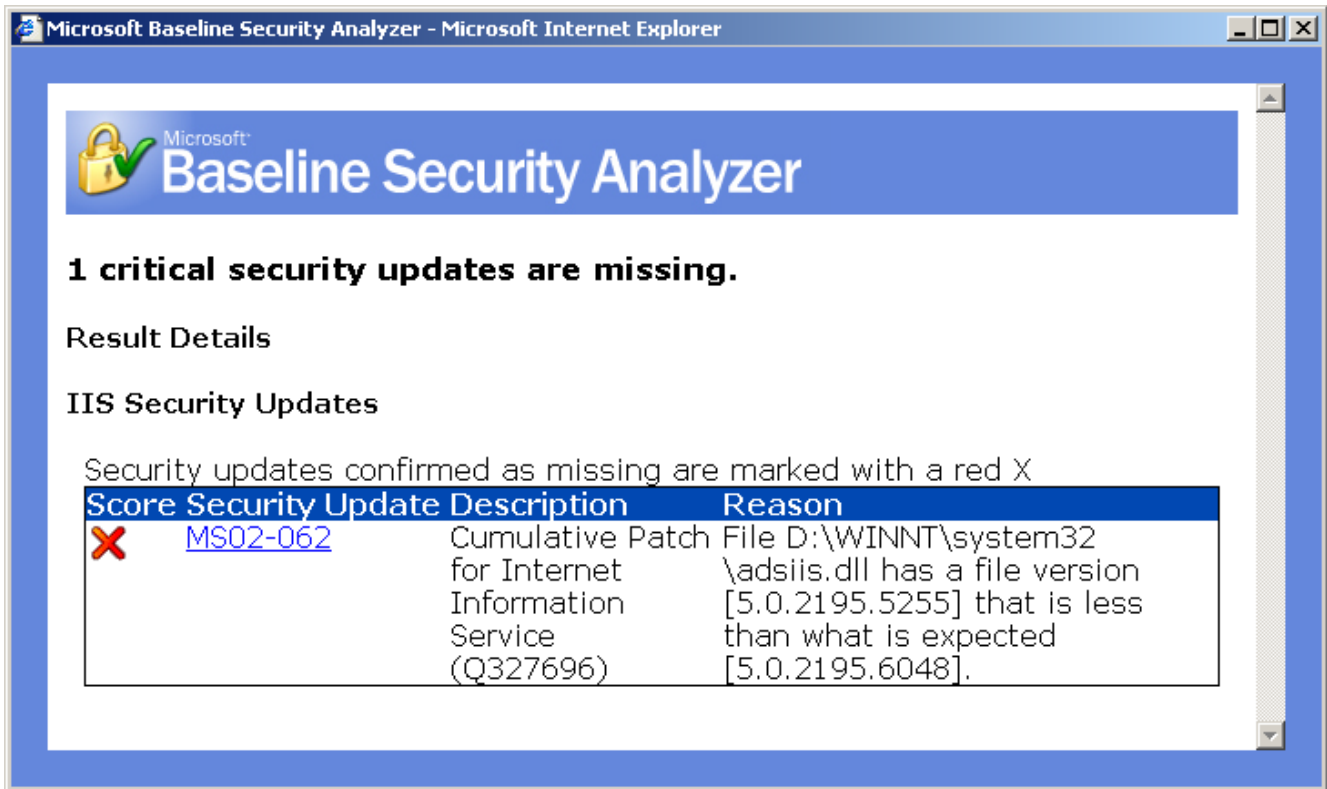


Figure 16-8: A List of Missing Security Updates for IIS is Displayed Under Result Details

Don't let your security policy rely on such inadequate scanning software and flawed publishing procedures. Assemble your own list of known good authentic hash codes of the most up-to-date Microsoft binaries (and software produced by other vendors), and use this set of trustworthy hash codes along with a simple hash verification tool instead. Before I show you how to build such a tool, it's important to look at one final aspect of MBSA that actually has some lingering residual value. It is part of the command line interface, and it's the successor to HFNETCHK.

Network Security Hotfix Checker (Formerly HFNETCHK)

The Microsoft Network Security Hotfix Checker was originally a command-line utility named HFNETCHK.EXE and is now an integrated feature of MBSA accessed through the following MBSA Command Line Interface (MBSACLI.EXE) parameter:

```
mbsacli.exe /hf
```

HFNETCHK itself derives from the original HFCHECK command-line utility that scanned for missing hotfixes in IIS version 5. HFNETCHECK was an enhanced version of HFCHECK designed to scan for missing hotfixes in IIS version 4, Windows NT 4, Windows XP/2000, Internet Explorer, and the SQL Server Developer Edition (MSDE) database server. The mssecure.xml file and its XML schema were designed originally for use with these hotfix scanning tools, and MBSA is the first enhanced security scanning tool produced by Microsoft to add new features on top of the base hotfix scanning capability afforded by mssecure.xml files. The XML schema for mssecure.xml is important because it attempts

to capture the complexity of hotfix and file version management challenges in the real world. The schema includes elements for listing the files supplied by each hotfix and the version numbers and checksums of each file. Registry keys applied by each hotfix installer that can help to determine whether that installer has ever been executed in the past are also included. Relationships between current hotfixes and previous hotfixes, such as dependencies and successor hotfixes that supersede other hotfixes or that undo (on purpose or by mistake) the changes made by previous hotfixes and service packs. All of this information and more is modeled in the XML schema for the mssecure.xml file.

Shavlik Technologies still creates updated versions of HFNETCHK and they offer a Pro edition with enhanced capabilities for enterprise-wide patch management. Figure 16-9 shows the output of the latest version of HFNETCHK that Shavlik made available for free download from their Web site well in advance of the outbreak of the Sapphire worm which hit unpatched SQL Server boxes on January 25, 2003. Note that Figure 16-9 includes no reference to the hotfixes released with MS02-039, MS02-056, MS02-061, or SQL Server Service Pack 3 that were critical to protect against Sapphire (a.k.a. Slammer) in spite of the fact that the latest version of the vulnerable file, ssnetlib.dll, was not present on the box. These hotfixes weren't listed in the output shown in Figure 16-9 (which was obtained just hours prior to the outbreak of the worm) because SQL Server was not installed on the IIS box being scanned. A manual review of Shavlik's version of mssecure.xml would reveal how critically-important it was to run HFNETCHK on every Windows box, and comparison of Shavlik's file with Microsoft's, cross-checking the output of MBSA with HFNETCHK, would have revealed that MBSA failed to detect the missing ssnetlib.dll at all even on vulnerable boxes that did have SQL Server installed, while HFNETCHK was able to detect the vulnerability.


```

Visual Studio .NET Command Prompt
D:\Program Files\Shavlik Technologies\HFNetChk>hfnetchk
Shavlik Technologies Network Security Hotfix Checker 3.86
Copyright (C) 2001-2002 Shavlik Technologies, LLC
Shavlik Technologies, LLC
info@shavlik.com (www.shavlik.com), 651-426-6624
All Rights Reserved

Please use the -v switch to view details for
Patch NOT Found, Warning and Note messages

Attempting to download the CAB from:
http://xml.shavlik.com/mssecure.cab

File was successfully downloaded.

Attempting to load .\mssecure.xml.

=====
Scan performed Thu Jan 23 17:24:37 2003
Shavlik Technologies Network Security Hotfix Checker, 3.86
Using XML data version = 1.1.1.589 Last modified on 1/22/2003.

Scanning WIN2KDEV
-----
Done scanning WIN2KDEV
-----
WIN2KDEV (172.16.0.1)
-----

* WINDOWS 2000 SERVER SP3

Patch NOT Installed      MS02-042      Q326886
Patch NOT Installed      MS02-045      Q326830
Patch NOT Installed      MS02-048      Q323172
Patch NOT Installed      MS02-051      Q324380
Patch NOT Installed      MS02-053      Q324096
Patch NOT Installed      MS02-055      Q323255
Patch NOT Installed      MS02-063      Q329834
Note                     MS02-064      Q327522
Patch NOT Installed      MS02-069      Q810030
Patch NOT Installed      MS02-070      Q329170
Patch NOT Installed      MS02-071      Q328310
Patch NOT Installed      MS03-001      Q810833

* INTERNET INFORMATION SERVICES 5.0 SP3

Patch NOT Installed      MS02-062      Q327696

* INTERNET EXPLORER 6 GOLD

Warning
The latest service pack for this product is not installed.
Currently Gold is installed. The latest service pack is Internet
Explorer 6 SP1.

Patch NOT Installed      MS02-058      Q328676
Patch NOT Installed      MS02-068      Q324929

* WINDOWS MEDIA PLAYER 7.1 GOLD

Patch NOT Installed      MS02-032      Q320920

D:\Program Files\Shavlik Technologies\HFNetChk>

```

Figure 16-9: HFNETCHK Should Be Used in Addition to MBSA

Because the original HFCHECK utility was for IIS version 5 only, it is not entirely clear that Microsoft intended for administrators of SQL Server boxes to use HFNETCHK at all for scanning their boxes. Since MBSA has a more generic name that clearly implies that the utility is for scanning of every Windows box, this miscommunication resulted in many people relying on MBSA rather than HFNETCHK, without understanding that MBSA was designed for warm fuzzy administrative reporting not real security scanning. There is no reason to trust HFNETCHK any more than you should trust MBSA, but there can be no doubt that the latest version of mssecure.xml always offers something of value to help you ascertain that your boxes are properly patched. Rely on HFNETCHK for hotfix scanning, but don't do so without manually reviewing mssecure.xml first. And don't consider your boxes to be proven secure, or even to be known to be running authentic Microsoft code, based on anything you see output by either scanning utility.

Both MBSA and HFNETCHK attempt to be user-friendly by making automated behind-the-scenes trust determinations, parsing mssecure.xml, and analyzing the code and configuration settings of your IIS box without cluttering up the screen with any extraneous technical details. By assuming that software is better at security configuration review and verifying trustworthy code and data than humans, Microsoft and Shavlik Technologies have created tools that exhibit the same self-verifying circular logic that plagues much of the Windows platform. While the MBSA and HFNETCHK software is essentially worthless (or at worst, dangerous) if you've already installed the latest service pack and properly hardened your IIS box, mssecure.xml itself is extremely important and valuable. With this one XML file, Microsoft gives you a standard digitally-signed means of receiving authentic communication from the source as to the availability of hotfixes for newly-discovered security vulnerabilities. Not only should you read this file every time it gets updated, you should use WinDiff or a similar file comparison tool to review every modification made between versions and you should also build your own security analysis tools that leverage the XML schema defined by Shavlik Technologies for Microsoft. The mssecure.xml schema will evolve as more Microsoft customers recognize the extremely important role of this type of digitally-signed structured one-to-many communication from vendors.

Hash Verification with Checksum Correlation

Hopefully by the time this chapter gets printed and bound in book form, the schema for mssecure.xml will have changed substantially. One area in particular is in dire need of a redesign and the word is that Shavlik Technologies is busy working on this very issue: replacing the insecure file checksums implemented in the original mssecure.xml schema with proper cryptographic hashes so that an attacker cannot easily create a malicious binary with the right filename, version number, and checksum such that MBSA and HFNETCHK will consider the malicious code to be the trusted authentic Microsoft code. As of this writing in February 2003, mssecure.xml still uses checksums rather than hash codes. This means that the only security analysis you're getting when you run HFNETCHK is a superficial review of hotfixes that may not have been installed yet on your IIS box. You can get this information yourself simply by downloading mssecure.xml manually and reading it with a text editor. Every system administrator keeps a written system administration log for each box that they manage, so hotfix installation status should never be an unknown unless version rollback has occurred by installation of conflicting hotfixes, service packs, or a system recovery. As a tool used one time to

establish a baseline patch status after a fresh OS build or a system recovery, MBSA is somewhat useful. MBSA should never be used as a security auditing file version and authentic code verification tool.

MBSA does not verify checksums based on the contents of mssecure.xml in spite of the fact that the file contains checksums. Likewise, Neither HFNETCHK (nor mbsacl.exe with the /hf parameter) do anything more than attempt to rationalize hotfix installation status based on infrequent use of these checksums. There is logic in HFNETCHK that makes use of checksums to resolve ambiguities between conflicting file version numbers caused by mistakes made repeatedly by Microsoft programmers when they recompile and rerelease files as part of a hotfix or service pack. When Microsoft compiles Windows source code to build a new version of a particular Windows binary, the programmers don't always remember to update the version number in the file's resource headers. This makes it impossible to determine based on file version number alone whether or not a particular patch is installed. When a file contains a newer version number than the one required by a particular hotfix, it is presumed that the newer version contains the fixes added to the earlier version so the checksum of an older version would naturally be ignored. Also, Microsoft issues hotfixes and service packs for reasons other than to patch security vulnerabilities, and these file updates are not tracked by mssecure.xml. On top of these complexities that limit the file analysis performed by Shavlik's tools, the use of checksums instead of cryptographic hashes makes it possible for attackers to replace authentic Windows binaries with malicious code that will produce the same checksum as the authentic binary when run through the MapFileAndChecksum API.

While the checksums found in mssecure.xml are easily spoofed, and routinely ignored in practice for the reasons cited above, the fact that they are present at all in mssecure.xml is better than nothing, especially if you'd like to write a simple security analysis tool of your own that looks specifically at whether or not the Windows binaries installed on your IIS box are likely to be authentic code published by Microsoft. Unfortunately, for reasons beyond current explanation, the checksums published in mssecure.xml simply do not match checksums computed in the wild using the same Win32 API functions as are used by Microsoft and Shavlik when the checksum value for each binary is populated inside mssecure.xml. The checksums of files found in the wild do match occasionally, but typically they will not match what is found in mssecure.xml even when MD5 hashes and SHA-1 hashes known to be the authentic hashes of particular Windows binaries are verified for these same files. This is a mystery for future explanation whose existence calls into question the accuracy and integrity of everything ever done with MBSA or HFNETCHK in the past.

Verifying hashes or checksums of Windows binaries is not sufficient to prove a box to be completely uncompromised. There are any number of ways for a Trojan or other nefarious tool to remain active or activate itself at boot time other than by replacing authentic Windows binaries with malicious rootkit binaries. Hooking and chaining techniques are a good example of this and the Applnit_DLLs sample created in Chapter 10 to countermand code execution relies on a well-known code injection feature built-in to the Windows platform that might be used maliciously. You must always verify the integrity of both code and data to ascertain that an IIS box is trustworthy. Registry settings are more commonly used for compromise than are rootkit Trojans. Verifying integrity of the

Registry is pointless if a rootkit is present on your IIS box. Both threat models are important to analyze in detail.

Authentic MD5 hashes of binaries published by Microsoft can be found in little-known and somewhat obscure files distributed by Microsoft as part of the hotfix and service pack process. With each hotfix or service pack, Microsoft supplies a temporary file used only during installation named UPDATE.VER. Inside each UPDATE.VER file is a single line for each Windows binary included in the update package. To verify MD5 hashes of your updated Windows binaries you must make a copy of these UPDATE.VER temporary files placed on the hard drive during installation. It can be tricky to copy these files before they are deleted, especially if the hotfix has no user interface that gives you a chance to switch to a command prompt or the Explorer in order to locate and copy this file. Once you have a collection of UPDATE.VER files, merge them together into one big file, allowing the newer MD5 hashes for the latest version of each file to replace any older or redundant lines for the same file. Then use code like the following that is designed to read an UPDATE.VER file and perform recursive analysis of every file in every subdirectory beneath the Windows root directory (or wherever else you might store forensic samples taken from a production box for the purpose of off-line hash verification) comparing the MD5 hash code contained in the UPDATE.VER file with the MD5 hash code computed for each file.

verifyMD5s: C# utility for MD5 hash verification

```
using System;
using System.Security.Cryptography;
using System.IO;
using System.Collections;
using System.Runtime.InteropServices;
namespace verifyMD5s {
class Class1 {
static HashAlgorithm md5Hasher = null;
static SortedList slUpdate = null;
static ArrayList alVerified = null;
static ArrayList alUnverified = null;
static ArrayList alVeriFailed = null;
static bool bShowUnverified = false;
static bool bGroupOutput = false;
[DllImport("Imagehlp.dll", EntryPoint="MapFileAndCheckSum")]
public static extern UInt32 MapFileAndCheckSum(String filename, ref UInt32 headersum, ref
    UInt32 checksum);
[STAThread]
static void Main(string[] args) {
DirectoryInfo di = null;
FileStream fsUpdate = null;
md5Hasher = MD5.Create();
slUpdate = new SortedList();
alVerified = new ArrayList();
alUnverified = new ArrayList();
alVeriFailed = new ArrayList();
```

```

try { fsUpdate = File.Open("update.ver", FileMode.Open); }
catch(Exception ed) {System.Console.WriteLine(ed);
return;}
StreamReader sr = new StreamReader(fsUpdate);
String s = sr.ReadLine();
String[] sSplit;
if(s.CompareTo("[SourceFileInfo]") == 0) {
try { while(sr.Peek() > -1) {
s = sr.ReadLine();
sSplit = s.Split('=');
slUpdate.Add(sSplit[0].ToUpper(),sSplit[1].ToUpper()); }}
catch(Exception e) {System.Console.WriteLine(e);}
sr.Close();
fsUpdate.Close();
if(args.Length > 0) {bShowUnverified = true;}
if(args.Length > 1) {bGroupOutput = true;}
di = new DirectoryInfo(Directory.GetCurrentDirectory());
recurseDirs(di);
if(bGroupOutput) { foreach(String entry in alVerified) {
System.Console.WriteLine(entry); }
foreach(String entry in alVeriFailed) {
System.Console.WriteLine(entry); }
if(bShowUnverified) { foreach(String entry in alUnverified) {
System.Console.WriteLine(entry + ": MD5 Hash Not Available"); }}}}}
static void recurseDirs(DirectoryInfo di) {
byte[] hash;
String sUFN, sFullPath, sOutput;
FileInfo[] fi;
UInt32 headersum, checksum;
String[] sSplit;
FileStream fs = null;
fi = di.GetFiles();
foreach(FileInfo f in fi) {
headersum = 0;
checksum = 0;
sFullPath = f.FullName;
try { sUFN = f.Name.ToUpper();
if(slUpdate.Contains(sUFN)) {
sSplit = slUpdate[sUFN].ToString().Split(',');
fs = f.Open(FileMode.Open, FileAccess.Read, FileShare.ReadWrite);
hash = md5Hasher.ComputeHash(fs);
fs.Close();
if(BitConverter.ToString(hash).Replace("-", "").CompareTo(sSplit[0]) == 0) {
if(MapFileAndChecksum(sFullPath, ref headersum, ref checksum) > 0) {
System.Console.WriteLine("CHECKSUM Failure"); }
sOutput = sFullPath + " (checksum=" + checksum + "): Verified MD5 Hash " + sSplit[0];
if(!bGroupOutput) {
System.Console.WriteLine(sOutput);
alVerified.Add(sFullPath); }

```

```

else { alVerified.Add(sOutput); }
if(f.Length.ToString().CompareTo(sSplit[2]) != 0) {
System.Console.WriteLine("WARNING: File Size Mismatch, possible MD5 collision on " +
    sFullPath); }}
else { sOutput = sFullPath + ": MD5 Hash Verification Failed " + sSplit[0];
if(!bGroupOutput) {
System.Console.WriteLine(sOutput);
alVeriFailed.Add(sFullPath); }
else { alVeriFailed.Add(sOutput); }}}
else { alUnverified.Add(sFullPath);
if(bShowUnverified && !bGroupOutput) {
System.Console.WriteLine(sFullPath + ": MD5 Hash Not Available"); }}}
catch(Exception ex) {System.Console.WriteLine(ex); }}
foreach(DirectoryInfo dir in di.GetDirectories()) {
if(dir != di) { recurseDirs(dir); }}
return; }}}

```

Notice that the C# code shown here uses `System.Runtime.InteropServices` to make the native Win32 API call to `MapFileAndChecksum`, the `Imagehlp.dll` function that computes the checksum of the body (excluding the headers) of a PE/COFF file using a SIP provider similar to the way that Windows File Protection works. The PE/COFF file format and SIP providers were discussed in Chapter 13 in the context of hashing, and the `MapFileAndChecksum` API is an example of these same methods being used to compute checksums. Again, the reason Microsoft likes to do things this way rather than checksum (or hash) the entire file is that localization, digital signatures, and other constructs are designed to be placed in PE/COFF header structures that aren't considered to be part of the executable code, or binary image, loaded into memory when a Windows binary is used at runtime. In the case of checksum computation there is another reason for using a SIP provider rather than compute a full-file checksum. The PE/COFF file format specifies that a checksum of a program file's binary image can be placed in the file's header to enable file integrity checking when the file is read into memory by the Windows loader. If the full file were checksummed instead, without using the SIP mechanism, then inserting the checksum after the file's checksum is computed would alter the file's checksum and the checksum embedded in the header would no longer verify the file's integrity. By manually comparing the output of this C# code against `mssecure.xml` you can correlate and compare checksums computed by the `MapFileAndChecksum` API with those computed by Microsoft or Shavlik and see for yourself that they rarely match.

The C# utility for MD5 hash verification shown here fills three `ArrayList` objects with the results of hash verification. It then displays these results in a different way depending on whether or not parameters are passed on the command line. Without parameters, the program sends to standard output the result of each hash verification attempt immediately after it occurs. With a single parameter of any value passed on the command-line the program also displays the full path of every file that it does not attempt to verify due to the fact that there is no MD5 hash present in the `UPDATE.VER` file used as input to the program. When two parameters are passed to the program, output is delayed until all file hashing is complete. The three `ArrayList` objects are then iterated and a complete list of files with verified MD5 hashes is displayed followed by a complete list of

files that failed MD5 verification and finally a complete list of files that were not scanned due to missing MD5 information.

Microsoft does not yet provide reliable, trustworthy utilities for verifying security of an IIS box. There can be no mistaking Microsoft Baseline Security Analyzer as anything but a smokescreen designed for political rather than security purposes. Use it and HFNETCHK from Shavlik Technologies with caution, but do use them to derive what little benefit they can provide. Then move on to real security analysis using proven computer forensics methods and your own source code that you know you can trust. When it comes to proving security for your IIS box, there is no room for error or deception, and nothing less than access to the source code of the tools you rely on for infosec forensic analysis will do. Appendix A of this book, or electronic copies of UPDATE.VER files that you locate from another trusted source, combined with a simple MD5 hashing utility like the one shown in this chapter, provide you with provable, reproducible, infosec forensics confirmation that your box has binaries installed that are the most current authentic binaries available from Microsoft.

To stay current with the most up-to-date list of known good authentic hashes of Microsoft's hotfix and service pack binaries, review mssecure.xml manually every time it changes and take the time to collect every UPDATE.VER file you receive with code updates. Authentic hash code verification is the real baseline security you should expect, and the minimum level of assurance you must demand. Once you have a list of the authentic hashes of the most up-to-date code you're supposed to have installed on the hard drives of your IIS boxes, you can verify this baseline security any time you wish using whatever infosec forensics techniques and tools you prefer. And you can also move on to the more difficult technical challenge of trying to determine if the data and the memory-resident code that are actually controlling the behavior of your IIS box are authentic instructions supplied by a trusted vendor or your own programmers.

