Enhancing Security of Unix Systems

Danny Smith
Australian Computer Emergency Response Team
c/- Prentice Centre
The University of Queensland
Qld. 4072.
D.Smith@auscert.org.au

Abstract

This paper examines the common threats to data security in open systems highlighting some of the more recent threats, and looks at some of the tools and techniques that are currently available to enhance the security of a Unix system. Since many programs are written without security issues in mind, the topic of secure programming methodologies is also discussed, with some examples of coding techniques that avoid security vulnerabilities.

1. Introduction

In November 1988 a "worm" program was released to the Internet, which (although not entirely intended by the worm's author) brought much of the network to a standstill [Spa88]. Many sites disconnected from the network until a solution could be found. This had an added disadvantage as many of the first fixes for the problem were distributed via the network.

In March 1991 a ship was lost in the Bay of Biscay. A weather forecasting satellite was not working as the integrity of a computer system at the European Weather Forecasting Centre in Bracknell, Berkshire had been broken by intruders[1]. The storm that caused the ship to sink had not been predicted [Aus93].

In mid-1993, a number of sensitive medical test results were changed from negative to positive by intruders. Various people around that country were then led to believe that they had cancer. It was up to Police to advise them that the results had been changed and were incorrect [Aus93].

Often when the topic of computer security is discussed, particularly when it relates to intruders gaining unauthorised access, a wide range of viewpoints can be expressed. Mostly the intruders do not cause harm, but when intruders perform acts like the sample described above, it is difficult to tell the difference between the "good" intruder and the "bad" intruder [Che92]. Therefore, these intruders must now be treated as criminals.

Over the past six years, computer vulnerabilities and the way they are exploited has changed character. In 1988, many intruders were simply exploiting poor password choices, poor system configuration, or software vulnerabilities [CER92], [Sto89]. However, in 1994, those types of attacks are still common, with the addition of people actively examining source code for operating systems and utilities in an attempt to determine any potential vulnerabilities. Network sniffing has become more common place.

This paper examines threats faced by system administrators despite their knowledge and diligence. It examines the common programming mistakes that allow exploitation of privileged programs by intruders. It overviews a number of tools available to assist in the detection of any exploitations, and then indicates a number of programming techniques required to prevent software vulnerabilities.

[1] This paper uses the term "intruder" to indicate a person who enters a computer system, or who accesses or modifies data, and does not contain the appropriate authorisation to do so. Other terms are used in the public domain such as "hacker" and "cracker", and these terms are continually debated for appropriateness. The term intruder is chosen here, as it indicates the criminal and antisocial intent of the activity, as viewed by the author.

2. Threats and Vulnerabilities

Many of the computer security problems experienced today relate to poor practices rather than software vulnerabilities. Tighter control of procedures can significantly reduce the number and severity of computer intrusions. There is however, a significant number of software packages that contain vulnerabilities that will allow an intrusion, despite the best procedures being implemented. Correct procedures will help to detect this class of intrusion much quicker, and reduce its impact.

This section of the paper examines the types of vulnerabilities that may still occur despite the actions and procedures of the organisation. As such, the class of activities that are generally under the control of a competent system administrator is not discussed.

2.1. Software Vulnerabilities

There are many techniques that a programmer must employ when writing privileged code (or at least, code that will run as the Id of another user) [Far91]. In essence, it is important to ensure that the user cannot control the environment in which the program executes, and that all error status returns are checked and handled appropriately. Failure to do this may result in the software integrity being compromised. These techniques are not reserved for privileged code, but are generally good coding practices. They include simple tasks such as initialising the operating environment to a known state, checking the status returns of all system calls, and parsing all arguments internally and not trusting a third party to do it correctly. The rest of this section is drawn from [Arn93], [Bis87], [GS91], and the analysis of recent software vulnerabilities.

2.1.1. IFS

One particular type of attack involves the IFS shell variable (Input Field Separator). This variable is used to indicate what characters separate input words to the shell. Whilst its functionality has been largely superseded, it lives on to cause unexpected results. For example, if a program calls the `system()` or `popen()` functions to execute a command, then that command is parsed by the shell first. If the user has control over the IFS environment variable, this may cause unexpected results. A typical scenario might be if the program executes the following code:

```
system( "/bin/ls -l ");
```

If the IFS variable has been set to contain the "/" character and a malicious program called "bin" is placed in the path of the user executing the program, then that program will be executed as the shell will have parsed the line as:

```
bin ls -l
```

which executes the program `bin` (in the current path) passing two arguments `ls` and `-l`. It is for this reason that a program should not get the shell to parse command lines by using the `system()`, `popen()`, `execlp()`, or `execvp()` commands to run some other program.

2.1.2. HOME

Another form of environment attack is the use of the `HOME` environment variable. Normally the `csh` and `ksh` substitute the value of this variable for the ~ symbol when it is used in pathnames. Thus if an attacker is able to change the value of this variable, it might be possible to take advantage of a shell file that used the ~ symbol as a shorthand for the home directory.

For example, if a shell file was referencing the `~/.rhosts` or `$HOME/.rhosts` file for the user running it, it is possible to subvert it by resetting the `HOME` environment variable before executing.

2.1.3. PATH

The `PATH` attack is characterised by the value and order of the file paths in the `PATH` variable. An inappropriate choice of path orderings may lead to unexpected results if a command is executed without reference to its absolute path. For example, consider the following `PATH` specification:

```
PATH = .:/usr/bin:/bin:/sbin
```

If someone had created a file called `ls` in a place that the current working directory is set to, then this would be executed in favour of the normal system `/bin/ls` command. If the file contained the following:

```
#!/bin/sh
```

```
(/bin/cp /bin/sh /tmp/.secret
  /bin/chmod 4555 /tmp/.secret)
  2>/dev/null
rm -f $0
exec /bin/ls "$@"
```

this would silently create a copy of `/bin/sh` which when executed would attain the identity of the person who executed the file. In addition, it would clean up any evidence and execute the real `/bin/ls` command so that the command would ultimately succeed without the user being aware that something else has happened.

In general, the use of relative path names either in referencing files or when executing programs should be considered as a poor programming practice.

2.1.4. Buffer Overflows

Software vulnerabilities also are made apparent through the use of poor programming practices. These practices may be performed through ignorance or inexperience, or the programmer may not care to do it correctly the first time.

A prime example of this was exploited by the Morris worm. The vulnerability was that the `gets()` system call was used which does not perform any length checking. Instead, the `fgets()` system call should have been used. This allowed a buffer to be overflowed under the user's control, and hence the program was forced to take inappropriate action. There are several other system calls that suffer this same fate, including `scanf()`, `sscanf()`, `fscanf()`, and `sprintf()`. Software system design may affect the use of `strcpy()`, `bzero()`, and `bcopy()` as well.

2.1.5. umask

Often, the value of `umask` (the default file protection mask) is set to something that is inappropriate. Many programs fail to check the value of `umask`, and often fail to specify a protection for any newly created files. Even if the program created a file, and then changed its protection to make it secure, a window of opportunity exists where an attacker could interrupt the program and possibly gain access to a writeable file. Therefore, it is important to establish a value for `umask`, prior to opening any files.

2.1.6. Status Returns

Another typical programming practice that can cause problems is the lack of making the effort to check the status returns on every system call. This means that if an intruder can gain control of the environment in which the program is running, they may cause a particular system call to fail, which the program blindly assumes will always work. This may cause the program to take further inappropriate action.

2.1.7. Catching Signals

Often a program fails to catch all the signals it possibly can, and react appropriately. This may allow an attacker to set their `umask` value to something inappropriate, and then send a signal to a privileged program, causing it to dump its core (some systems allow this). When this happens, the core file is owned by the effective UID of the running program, but protected with the value specified by `umask`.

2.1.8. Array Bounds Checking

A recent vulnerability in `sendmail` allowed a large integer to be passed in as a parameter. This integer was used as an array index. Since the number was so large, it was actually treated as a negative number by the computer, allowing data to be overwritten further back in the program. This allowed privileged access as a result.

The program handled large positive numbers but was not equipped to handle negative numbers. Changing the definition of the variable from `int` to `unsigned int` was all that was required.

2.2. Examining Source Code

Since there are now known coding practices to avoid when writing programs (particularly ones that will run in a privileged mode), experience suggests that attacks are being launched at particular programs, usually after careful examination of the source code. In general, the source code is freely available and this allows anyone to examine it looking for potential flaws. This was not the usual form of attack up to six years ago, but it is becoming more common place these days [CER92]. In some aspects, this will result in some good as the programming errors will be finally rectified, and programming methodologies will become better understood.

2.3. Trojan Horses

Trojan Horses are named after the legend of the same name. Computer style trojan horses resemble normal programs that a user wishes to run, such as editors, login programs, or games. While the program may be appearing to do what the user wishes it to do, it is actually doing something completely different (such as deleting files, storing passwords for later use, reformatting disks). By the time the user is aware of a problem, it is far too late [KC90].

Trojan Horses can appear in many different places. They can be found inside programs that have been compiled, or in system command files executed by system administrators. Other forms of trojan horse include sending commands to people as part of a message (such as electronic mail or a message to a terminal). Some mail handlers allow the user to escape to the shell and execute commands. This feature can be activated when the message is read. Sending a particular message to some terminals can have a command sequence stored in the terminal, and then have that command played back as though it was typed on the keyboard. Editor initialisation files are also a favourite place for storing trojan horses.

Trojan horses are unfortunately, very common [GS91]. As soon as a system is compromised, the attacker usually modifies the system in many ways to ensure that if the original intrusion is discovered, they can always get back into the system. This has the added effect of raising the cost of recovering from a compromise, as the *entire* system must be painstakingly checked for the presence of any Trojan Horses.

2.4. Network Monitoring and Data Capture

One of the threats now being faced by computer systems is the ease at which data can be captured while it is being transmitted between computers [Bro93]. In the past when large central systems were the norm, this did not pose a large threat, but the advent of heterogeneous computer networks that span the globe mean that sensitive data may travel beyond the control of an organisation. There are many packages available that can be used to monitor data as it passes by on a network link (for example, [MLJ92]). Particularly vulnerable are the bus style networks (such as Ethernet) where data destined for a particular host can be viewed by any host connected to the network.

This now means that any data can be captured and used for different purposes. This does not only include sensitive data, but may also include protocol exchanges (such as a login sequence, including the password). Authentication sequences are one of the most vulnerable exchanges as they are the critical decision point when granting access privileges.

Data does not always have to be captured from the network itself. By installing a Trojan Horse in the network software or the application, data may be captured and saved on the disk for later examination. This may defeat the standard defence of encrypting data that is to be transmitted across the network [Bro93], [Din90].

2.5. Software Interaction and Configuration

Ultimately, the underlying reason for security problems can be reasoned to stem from the fact that computer systems and the software that run them are becoming more complex each day. Given that generally no single person writes the entire system, it is impossible to predict the interaction of several components of a system, especially at the border conditions and in obscure error cases. A recent example of this was a problem with `/bin/login` accepting invalid parameters from a number of other programs.

As well as the complexity of software interaction, programmers are giving the system administrator a huge range of choices. The job of configuring a system is becoming so complex that simple errors may lead to subtle security problems. There is a trend towards electronic testing of computer systems security; evaluating the ability to penetrate the system through the use of programs, both external and internal to the system itself [CER93], [Gro93], [Kur90]. This however only reports on *known* vulnerabilities, and does little to detect new vulnerabilities.

2.6. Concluding Remarks

System intruders are definitely becoming more sophisticated. The average age of the intruder is increasing, and law enforcement are no longer dealing with wayward teenagers. Law Enforcement are now arresting alleged intruders that have graduated with First Class Honours degrees [Aus93]. This is also reflected in the types of attacks that are being witnessed over time.

In 1988, the majority of computer system intrusions resulted from exploiting poor passwords, and system vulnerabilities.

In 1994, the techniques of six years ago are *still* in common use (and are *still* successful!). However, intruders are now exploiting protocol weaknesses in attempt to fool servers into performing some service, there is more network sniffing looking for valuable information, and many of the intruders, when arrested, possess system source code most likely with a view of examining it for more flaws or using it to insert Trojan Horses.

As the sophistication of intruders grows, so does the sophistication of the system administrators *and their tools* need to grow.

3. Available Tools

A number of tools and techniques are available to help the system administrator and system programmer with their task. This section includes a selection of them with a discussion on what they do. Whilst these tools do not prevent software vulnerabilities, they may help detect any intrusions that may occur through the exploitation of those vulnerabilities, or prevent the use of network sniffers to capture important authentication data.

3.1. Cryptographic Tools

3.1.1 Kerberos

The following analysis is drawn from [Ste90], [BM91], and [KCS90].

The Kerberos authentication system was produced at MIT as a part of Project Athena. It is a system that uses protocols which allow authentication to take place, even under the assumption that the network is under the complete control of an enemy. Kerberos uses a private key cryptosystem to protect the information from disclosure and modification. The user interface is the same as that for normal passwords.

The major strength of Kerberos is that the password is never transmitted on the network in plain text. This reduces the likelihood of the password being captured and replayed.

The tickets and authenticators include a timestamp which aids in preventing replay attacks (where an intruder replays a valid authentication sequence).

This style of authentication was designed with the distributed or networking environment in mind. It is well suited to the client-server model often used in networking applications.

Since both the ticket and authenticator contain the network address of the client, another workstation cannot use stolen copies without changing their network address.

Kerberos also contains a number of minor deficiencies which should be well understood use it effectively. Installing Kerberos will increase the level of security over normal passwords, provided its limitations are understood and accepted.

The timestamps are critical to the successful operation of Kerberos. The times on the source and target machines must be closely aligned, or it will be possible that a valid ticket will be rejected as fraudulent. Typically, a clock drift of five minutes will cause a denial of service.

Relying on the time to be in synchronisation means that one should also protect the protocols that set the time, so that an enemy cannot adjust the time to their will via this mechanism.

Tickets are reusable but have a lifetime. After having been authenticated for a long period (typically eight hours), it is necessary to generate a new ticket by entering the login name and password to Kerberos again.

Within the Project Athena environment (and hence, Kerberos), the primary need is for user to server authentication. When a user accesses a workstation, they need access to private files residing on a server. The workstation itself has no such files, and hence has no need to contact the server or even identify itself. This contrasts with a typical UNIX system's view of the world. Such systems do have an identity, and they do own files. Assorted network daemons transfer files in the background, clock daemons perform management functions, electronic mail and news is transferred, and so on. If such a machine relied on servers to store its files, it would have to prove its identity when talking to these servers. Kerberos is not a peer to peer system, intended for use by one computer's daemons when contacting another computer.

In a workstation environment, it is quite simple for an intruder to replace the login command with a version that contains a trojan horse (captures accounts and passwords). Such an attack negates the primary strength of Kerberos, that passwords are not transmitted in plain text over a network. While this problem is not restricted to Kerberos environments, the Kerberos protocol makes it difficult to employ the standard countermeasure: one-time passwords.

The authenticator relies on the use of a timestamp to prevent against replay. Given that the lifetime of an authenticator is typically five minutes, a window of opportunity exists where a stolen live authenticator could be used to fraudulently gain access to a server. It has been suggested that the proper defence is for the server to store all live authenticators so that a replay could be detected. However, on UNIX systems, TCP-based servers generally operate by forking a separate process to handle each incoming request. Since the child and the parent do not share any memory, it is not convenient to communicate to the parent (or any other child processes) the value of any authenticator that is presented. UDP-based query servers generally use a single process to handle all incoming requests, but may have problems with legitimate retransmissions of the client's request if the answer gets lost.

Whilst the Kerberos system guards against having to send the password in plain text, the passwords chosen are much the same as the standard UNIX password, which suffer the same fate of normal passwords in a password guessing attack. The encrypted password is not freely available (it is stored on the Kerberos server), so to succeed with such an attack would require the password to be guessed, rather than just replaying the authentication sequence again.

Tickets are based upon a system's IP address. On multi-homed systems (systems with more than one network interface and IP address), this may cause a problem as the ticket will only be valid through one of those interfaces.

Tickets are stored in `/tmp` which does not work very well for multi-user systems.

3.1.2 DES

One of the most widely used encryption systems today is the Data Encryption Standard developed in the 1970s by IBM [FIP77], [CLS91]. DES is a bit permutation, substitution, and recombination function performed on blocks of 64 bits of data and 56 bits of key (8 characters of 7 bits). The algorithm is structured in such a way that changing any bit in the input has a major effect on almost all of the output bits.

The DES algorithm can be used in four modes:
Electronic Code Book (ECB);
Cipher Block Chaining (CBC);
Output Feedback (OFB);
Cipher Feedback (CFB).

Each mode has particular advantages in some circumstances, such as transmitting data over a noisy channel, or when it is necessary to decrypt only a portion of a file.

DES uses the same key to encrypt the data and decrypt the data. Therefore, it is essential to use techniques that keep the secrecy of this key intact. Practical experience of using DES in a global situation highlights the difficulty of using DES in groups where keys must be distributed regularly to differing timezones. Poor key management leads to the reduced effectiveness of DES.

3.1.3 MD2, MD4, MD5

The MD2 Message Digest Algorithm [RFC1319] was created as part of the Privacy Enhanced Mail package The MD4 Message Digest Algorithm [RFC1320] was designed to exploit to 32-bit RISC architectures to maximise its throughput, and does not require large substitution tables. The MD5 Message Digest

Algorithm [RFC1321] is a proposed data authentication standard. MD5 attempts to address potential security risks found in the speedier but less secure MD4.

The message digest algorithms generate a 128-bit signature (fingerprint or message digest) from a given block of text. The signature is designed to prevent someone from determining a valid block of text from a given signature or to modify a block of text while keeping the same signature.

3.2. Security Assessment Tools

3.2.1 Tripwire

Tripwire is a file integrity checker using a number of cryptographic checksumming algorithms in parallel for added security [KS92]. Algorithms such as CRC-16 and CRC-32, commonly used to checksum data packets for transmission across a network [Tan89], do not provide sufficient strength to protect the integrity of files against a determined intruder. There are public domain tools that will help to "recreate" a valid checksum on files, while still maintaining file size. This is especially true of system binaries.

Tripwire makes use of several message digesting algorithms. These are:

MD5
MD4
MD2
Snefru
CRC-32
CRC-16

The use of more than one of these algorithms in parallel greatly decreases the chances of an intruder being able to modify a monitored file without detection. Initially, a reference database is built, immediately after the installation of the operating system and any products, and prior to reconnecting to the network. This way, one can be sure that the files have not been modified by an intruder. The output of Tripwire (as well as Tripwire itself) should be kept on a hardware write protected disk to prevent modification (a read-only mounted partition is not sufficient as this may be remounted read-write by the intruder). Tripwire should then be run at regular intervals to verify the integrity of key system files. Another alternative to using hardware protected media is to print out a copy of Tripwire's results. An intruder must gain physical access to the premises to adjust the original data from Tripwire. This helps if there is any suspicion on the integrity of the Tripwire database.

It is meaningless to use Tripwire to protect a file such as the system password file as users have the ability to change their password at any time, and thus the file checksums will also change.

3.3. Security Enhancement Tools

3.3.1 TCP Wrapper

TCP Wrapper (also known as LOG TCP) is a package that is used to monitor incoming IP connections, log them, and provide a number of add-on services including a limited form of access control and some sanity checks [Ven92].

The first function is to log connections. Any connection to an IP service that has TCP Wrapper enabled for it will write a connection record to the `syslog` daemon, containing the time and the source of the connection.

If the access control has been enabled, the list will be checked to see if the source of this connection has been allowed or denied access to that IP service. If the service is denied, the connection is aborted. If the service is allowed, then the normal daemon is executed.

If the name checking has been turned on, the wrapper will verify that the name to address mapping is the same as the address to name mapping. If there is any discrepancy, the wrapper concludes that it is dealing with a host that is pretending to have someone else's name (as in an attack on the "r" commands). If this is detected, it is logged and the connection aborted.

TCP Wrapper is an extremely simple, and yet effective tool. It is very useful in preventing connections from outside an organisation from approaching the systems. It is possible to allow certain connections (for example, mail) to the systems, while restricting others. Even if an intruder learns an account and password for the system, they must first penetrate a "trusted" system before they can gain access to the system [Cur90]. It is therefore imperative that users do not use the same password on all systems.

The TCP Wrapper, when properly configured, will reduce a system's exposure to intruders, and hence reduce their ability to compromise the security of a system by exploiting software vulnerabilities.

3.3.2 Token Generators

Token Generators are hardware packages that implement password "tokens", or one-time use passwords [Bra90], [CER92], [Ell92]. Token generators are implemented using a variety of schemes.

One system operates by challenging the user with a seven digit number (in phone number format). A PIN number and the challenge number are entered into the hand held device, and it gives a seven digit response code to reply with.

Other systems use a changing, non-reusable password system. Each time a user authenticates, a new password is supplied by the hand held device. There is no challenge-response system, and the user must keep in synchronisation with password usage to prevent a denial of service. Some systems can support single use password generation for up to eight separate host systems. Some of these systems require the user to enter a PIN before the next password is issued.

Another system displays the password continuously, changing it every minute or so. The host must not only keep the user's key (for generating the same sequence), but also a synchronised clock.

The one-time password system is extremely effective in preventing replay attacks, provided the enemy does not know the sequence of generated passwords (either by guessing, or possession of a similar device and key).

One of the major disadvantages is that to authenticate, a user must carry the hardware with them at all times. If they do not possess their hand held device, then authentication cannot take place.

Some of the systems have a requirement for synchronised clocks. This may cause the system to suffer a denial of service due to clocks slipping, or an attacker targeting the clock synchronisation protocols to set the time to any desired value.

3.3.3 S/Key

S/Key is a software system designed to implement a secure one-time password scheme [KHW93]. It uses 64 bits of information transformed by the MD4[2] message digest algorithm [RFC1320]. The 64 bits are supplied by the user in the form of six English words that are generated by a secure computer. Ultimately, this computer could be a pocket sized smart card, a standalone PC or Macintosh, or a secured machine at work.

The system forms a starting key by passing several items of information (including a secret password) through MD4. The starting key is then processed through MD4, and the resulting 128 bit signature collapsed to 64 bits. This 64 bits is passed back into the MD4 function, and again the result collapsed to 64 bits. This continues until the desired range of passwords is reached. For example, if the user requires passwords 95 to 99 be generated, the one way function is performed 95 times, and then the results for the iteration number 95, 96, 97, 98, and 99 are displayed. These form the one time passwords. The next password to be used is the one with the highest unused number. In this example, that would be password number 99. Note that password number 99 was generated by passing password number 98 through MD4, and collapsing the output down to 64 bits. Therefore, it is not possible to determine password number 98 (the next password to be used) from knowledge of password number 99. If the user does not correctly enter the secret password, a number of one-time passwords will still be generated, but they will not be valid.

The 64 bit passwords are processed through a routine that maps set bit positions onto small English words. This allows for an easy display of the passwords for the users to interact with. For example, a password may look like:

```
SIT LEFT FLEW MALT MEL PUN
```

---

[2]S/Key was written using MD4 as MD5 was not available at the time. Versions of S/Key using MD5 are now available, but currently are not in widespread use.

Passwords are displayed in this form, and are entered by users in this form. They are then converted back to the 64 bit representation for comparison.

The S/Key system provides a simple yet effective solution to the problem of intruders monitoring a network for passwords.

There are also replacement programs for the `login` and `su` programs which prompt for the one-time password with a challenge. Sample password generation output looks like:

```
host# key -n 3 99 sh42277
Enter secret password:
97: MASK THIS WART RUE ANNA IRON
98: TALL TOY CALF AWN HOOK LIT
99: SIT LEFT FLEW MALT MEL PUN
host#
```

Now when a user wishes to use the one-time passwords, the following happens:

```
host> su
s/key 99 sh42277
Password:
!enter SIT LEFT FLEW MALT MEL PUN
host# ^D
host> su
s/key 98 sh42277
Password:
!enter TALL TOY CALF AWN HOOK LIT
host#
```

If the wrong password sequence is entered, it is treated the same as an incorrect password.

4. Programming Techniques

Whilst some of the problems with security on computer systems are related to design, the proliferation of third party software packages has opened up a new world of security vulnerabilities. Often the problems relate to either inexperienced programmers or inadequate care when coding the system. The problem of how to write secure systems has been analysed for many years, and it is possible to write secure programs, if a number of basic mistakes are not performed.

Many of the solutions and programming styles simply come down to being as conservative as possible with programming, and never trusting the environment the program is operating in. The points detailed here are taken from [Bis87], [Far91] and the analysis of software vulnerabilities.

4.1. File descriptors

All unnecessary file descriptors should be closed before calling `exec()`. `exec()` has a documented feature than when the new program is called, "*descriptors open in the calling process remain open in the new process*" [Sun90b]. This means that if the first program was reading a sensitive file using privileges, and calls a user program via `exec()` without closing the file descriptor to that sensitive file, then the user program will also have access to that file.

4.2. Process environment

The *entire* environment in which the process will be run should be verified and reset by the program. This may involve setting known values in environment variables like `HOME`, `PATH`, and `IFS`, setting a valid `umask` value, and initialising all variables.

Simply completely zeroing the environment may not be effective, as other programs that are called by `exec()` may require these variables to function. Such variables may be `USER`, `SHELL`, and so on.

4.3. Filenames

A typical mistake is when writing shell scripts is to write them the same way a user usually interacts with the shell. Users generally use relative filenames for program and filenames specifications. An example of this might be when trying to remove a file, the user will type:

```
rm filename
```

rather than:

```
/bin/rm filename
```

If relative filenames are used to specify programs to execute, this leaves the shell script open to attack by inserting an `rm` program higher in the `PATH` than `/bin`.

This is closely related to the previous problem of determining the environment. Relative filenames and program names allow an intruder to determine the exact location where the file will be read or written to, through control of the environment.

4.4. Signals

When a process dumps core (terminates abnormally taking a copy of the memory image onto disk), the owner of the core file is the same user identity as the effective user identity of the running process. If this process is running as a SUID process, the user identity may be different to the person running the program. If the `umask` value has been set to something inappropriate, then it may be possible to write over the `core` file, while maintaining its original ownership. However, there may be race conditions introduced when catching signals. Signals can be safely ignored, but it may not always be appropriate to do this.

The use of signals to force premature termination of a process is not always obvious., and its success depends on the design and execution flow of each particular program.

4.5. Error recovery

A privileged program can never assume that all operations will succeed due to its privileged status. Subtle errors may occur that were never expected (such as not being able to access a file, a full disk device, or running out of file descriptors). These errors must be handled correctly. Recovery should not be attempted unless the recovery is guaranteed. Once a program loses control of its environment, it may be easy to force it to perform inappropriate actions.

4.6. Input data

Data should be bounded, and verified for syntactic correctness, integrity, and origin if possible. Storing input data in a protected file is not sufficient grounds for waiving the responsibility for verifying it. Any input data under the direct or indirect control of a user is particularly risky, and should always be treated with the utmost of suspicion. A common mistake is to assume that since one program wrote the input data, it will always have the correct format and be valid. Data may be created through exploitation of another privileged program, which when combined with this attack grants the intruder privileged access.

4.7. Race conditions

An example of this might be that the program creates a file, and in the next command protects it. In between the two commands, a window of opportunity may exist for an intruder to gain access to a poorly protected file that allows them to gain further access to the system. Attacks against race conditions (once identified) are generally automated, thus allowing the window of opportunity to be explored at a much greater rate than by hand. The program state should never be left vulnerable, not even for a single instruction.

Several attacks of this nature have recently been discovered and reported. These include programs such as `passwd` and `mail`. Many of these attacks involve switching symbolic links to files during a window of opportunity, forcing the program to act on the incorrect file.

4.8. Programs changing UID

Rather than letting a privileged program run with privileges all of the time, many programmers use the privileged status to perform the functions requiring it, and then change the effective user identification of the running program to be the same as the user that is executing it. In this way, the program can now only access system objects using the privilege of the executing user.

If the program is such that it must reacquire privileges periodically to perform a privileged function (available on some Unix systems), it is possible for a user to gain control of the running image whilst it is unprivileged, and then maintain control of the program when it changes to privileged mode.

One solution to this problem is to have the program execute as the user always, and to call a privileged program to perform only the privileged functions. This must be very carefully performed, as the user will have control over the environment and parameters that are passed to the privileged program.

4.9. Permissions

A recent program vulnerability occurred because a privileged program did not check the file's permissions before attempting to open it. The filename was specified by the user, and since the program was running in a privileged mode, the open would always succeed. Initially, this resulted in allowing any file on the system to be read by the user.

A further complication of this error involving a race condition allowed the user to gain privileged access.

4.10. File types

A number of vulnerabilities have resulted from the use of information stored in the `/etc/utmp` file. This file maintains a list of the current users on the system. Several programs use this file to identify users that should be advised of certain events on the system. The file identifies which terminal the user is logged in on. To allow write access to a protected terminal owner by another user, the various programs are required to be privileged.

Unfortunately, many of these programs do not test to see if they are actually writing to a terminal or not (since all devices look just like a normal file). By careful modification of the `/etc/utmp` file (which is writeable on some systems), it is possible to write over any file on the system, and thus gain privileged access.

4.11. Dynamically linked libraries

Some operating systems make use of dynamically loadable libraries [Sun90a]. When the image is activated, the library is loaded with the image from a path that is optionally controlled through the use of an environment variable. It is possible to replace a system library with a user created library. SUID programs will only load libraries from a fixed set of trusted library areas, but if the SUID program calls a non-SUID program while still running privileged, then any system functions that are called by the non-SUID program may have been created by the user to perform tasks other than those intended by the system designers. Once again, careful control of the environment is essential to prevent this type of attack.

4.12. chroot

A vulnerability existed in a version of `ftpd` recently which allowed arbitrary commands to be executed as `root`. This occurred despite the program being configured to operate in a `chroot`ed environment [Sun90b]. Files that exist outside of the `chroot` space that are required by the program must be accessed before the call to `chroot()`. For example, to perform valid user authentication, files like `/etc/passwd` must be accessed. The vulnerability occurred prior to establishing the restricted environment.

5. Conclusion

The problem of computer and network security is an extremely complex one. It has received increased attention over the past six years, and many tools and techniques have been developed to combat a wide range of threats. In particular, more attention is being devoted to the avoidance of common programming errors, and developing techniques that avoid well known vulnerabilities.

6. Bibliography

[Arn93] Arnold N., *UNIX Security: A Practical Tutorial*, McGraw-Hill Inc., 1993.

[Aus93] Austen J., Fifth Computer Incident Handling Workshop, St. Louis, MO., August 1993.

[Bis87] Bishop M., *How to Write a Setuid Program*, ;login, Volume 12, Number 1, January/February 1987.

[BM91] Bellovin S. and Merritt, M., *Limitations of the Kerberos Authentication System*, Proceedings of the USENIX Winter 1991.

[Bra90] Brand R., *Coping with the Threat of Computer Security Incidents: A Primer from Prevention through Recovery*. CERT 0.6, June 1990.

[Bro93] Brown L, *On Implementing Security Extensions to the TCP Transport Layer*, Proceedings of the 16th Australian Computer Science Conference (ASCS-16), Brisbane, February 1993.

[CER93] Computer Emergency Response Team Advisory 93:14, *Internet Security Scanner (ISS)*, September 1993.

[Che92] Cheswick W.. *An evening with Berferd in which a Cracker is Lured, Endured, and Studied*, Proceedings of the Winter USENIX Conference, San Francisco, January 1992.

[CLS91] Caelli W., Longley D., and Shain M., *Information Security Handbook*, Stockton Press, 1991.

[Cur90] Curry D., *Improving the Security of your UNIX System*, ITSTD-721-FR-90-21, SRI International, April 1990.

[Din90] Dinkel C., *Secure Data Network System (SDNS) Network, Transport and Message Security Protocols*, NIST, NISTIR-90/4250, March 1990.

[Far91] Farrow R., *Unix System Security: How to Protect your Data and Prevent Intruders*, Addison-Wesley, April 1991.

[FIP77] Federal Information Processing Standards Publication 46, *Data Encryption Standard*, National Bureau of Standards, U.S. Department of Commerce, January 1977.

[Gro93] Grottola M., *The UNIX Audit: Using UNIX to Audit UNIX*, McGraw-Hill Inc., 1993.

[GS91] Garfinkel S. and Spafford G., *Practical UNIX Security*, O'Reilly and Associates, Inc., 1991.

[KC90] Kaplan R., and Clyde R., *Viruses, Worms, and Trojan Horses - Part VI: The War Continues*, Proceedings DECUS Fall 1990, Las Vegas, 1990.

[KCS90] Kohl J., Neuman B., and Steiner J., *The Kerberos Network Authentication Service*, MIT Project Athena, Version 5 Draft 3, October 1990.

[Kur90] Kuras J., *An Expert Systems Approach to Security Inspection of UNIX*, Proceedings of the UNIX Security Workshop II, Portland, August 1990.

[RFC1319] Kaliski B., *The MD2 Message-Digest Algorithm*, Network Working Group, RFC1319, April 1992.

[RFC1320] Rivest R., *The MD4 Message-Digest Algorithm*, Network Working Group, RFC1320, April 1992.

[RFC1321] Rivest R., *The MD5 Message-Digest Algorithm*, Network Working Group, RFC1321, April 1992.

[Spa88] Spafford, E., *The Internet Work Program: An Analysis*, Technical Report CSD-TR-823, Department of Computer Science, Purdue University, November 1988.

[Ste90] Stevens W., *UNIX Network Programming*, Prentice Hall, 1990.

[Sto89] Stoll C., *The Cuckoo's Egg*, Doubleday, 1989.

[Sun90a] SunOS Reference Manual, Volume 1, SUN Microsystems, Revision A, March 1990.

[Sun90b] SunOS Reference Manual, Volume 2, SUN Microsystems, Revision A, March 1990.

[Tan89] Tanenbaum A., *Computer Networks*, Prentice-Hall International Inc.1989.

7. Other Information Sources

[CER92] Computer Emergency Response Team, *Internet Security for UNIX System Administrators*, Presented at AARNet Networkshop, December 1992.

[Ell92] Ellison C., *RESULTS: challenge login devices*, Usenet newsgroup sci.crypt, 6 October 1992.

[KHW93] Karn P., Haller N., and Walden J., *S/Key One Time Password System*, anonymous ftp from thumper.bellcore.com, July 1993.

[KS92] Kim G. and Spafford E., *README file from Tripwire system*, anonymous ftp from cert.org, November 1992.

[MLJ92] McCanne S., Leres C., and Jacobson V., *README file from tcpdump system*, anonymous ftp from ftp.ee.lbl.gov, May 1992.

[Ven92] Venema W., *BLURB file from TCP Wrapper system*, anonymous ftp from cert.org, June 1992.