A Short Course On

# Systems Administration and Security Under Unix

by Dr. Frederick B. Cohen

ISBN# 1-878109-13-8
ASP Press
PO Box 81270
Pittsburgh, PA 15217
USA

# Contents

# Chapter 1

# You Manage Resources!

We take the philosophy that you administer a computer system or network by managing its resources. You have a set of resources with various capabilities, and as the administrator, you decide how to allocate those resources to optimize your operation.

In order to administer a computer system effectively, you have to know what the resources are, what their capabilities are, and how they interact with each other. That is the main purpose of this course; to describe the available resources under UNIX$^{tm}$, to discuss the capabilities they provide and how UNIX$^{tm}$ accesses those capabilities, and to consider how their interactions affect operational efficiency.

## 1.1 Physical and Logical Resources

There are two different types of resources in modern computer systems; physical resources and logical resources. Physical resources typically include processors, memory, and peripheral devices. Logical resources are operating system abstractions that are given temporary control over physical resources.

Physical resources vary fairly dramatically from computer to computer. For example, a typical PC system might have 640K of memory, one 20Mbyte Winchester disk, one floppy disk drive, a single keyboard, and a single video display. A typical mainframe system has several parallel processors, hundreds of disks, tens of millions of bytes of memory, hundreds of terminals, tapes, and other special purpose peripherals, and is connected to a global network with thousands of other similar computers.

Logical resources are essentially identical on all UNIX$^{tm}$ systems, and are generally divided into *processes* and *files*. One of the major advantages of UNIX$^{tm}$ over other systems from a programming, administration, and usage standpoint, is that the wide variety of physical resources can be controlled through these two simple abstractions. The net effect is that UNIX$^{tm}$ administration is not fundamentally different for a PC than for a mainframe. It's only a matter of scale.

We sometimes use the term "space" to talk about *files*, and the term "time" to talk about *processes* because these are the basic abstract types of resources available in a computer, and they are the fundamental performance related phenomena that we are concerned about in dealing with computers from an optimization standpoint. Time can generally be traded for space in using computers, and this is the major performance related tradeoff that we have to control in order to efficiently use the resources of a UNIX$^{tm}$ system.

## 1.2 Users

Timesharing systems like UNIX$^{tm}$ are designed to allow multiple *users* to contemporaneously invoke multiple *processes* using the same hardware. In the early days of timesharing, *files*, *processes*, and *users* simply coexisted without any protection, but as a result, systems were very unreliable, and *users* often lost all of their information due to an error in another *user*'s program. Eventually, operating systems were changed so as to protect *users*, *processes*, and *files* from each other, so that each relates to a virtual computer that is independent of other factors in the environment to a large degree.

## 1.3 Security

As a field, security encompasses a wide variety of techniques and principles, and includes such areas as physical security, personnel security, computer security, legal issues, and cryptography. Our concern is considerably more focussed. We are only concerned with three security objectives in our discussions; privacy, integrity, and availability.

- Privacy is protecting a person from undesired disclosure of information.
- Integrity is protecting a person from undesired corruption of information.
- Availability is protecting a person from denial of deserved services.

Notice that when we talk about security objectives, we talk in terms of people. That is our major concern; assuring that people can do what they are supposed to do and can not do what they are not supposed to do. In UNIX$^{tm}$, people roughly correspond to *users*, and act through *processes*. *Processes* interpret *user* keystrokes, and act for the *user* in manipulating *processes* and *files*.

## 1.4 Program Execution

Under UNIX$^{tm}$, "programs" are "executed" by *processes*, either by the *process* interpreting information, or by a *process* replacing its image in memory with the binary program image stored in a *file*. In UNIX$^{tm}$, a search *path* is used to find the program to be executed. The search *path* is stored in an environment variable called *PATH*, which consists of a sequence of directory names separated by ';'s. Each of the directories named in the sequence is searched in order for an executable *file* with the right name. Once found, the *file* is interpreted.

# Chapter 2

# Users and Groups

*Users* are identified by *user* identifications (UIDs), each of which is associated with an integer in the range of 0 to a system specified maximum. We will use UIDs and the integers associated with them interchangeably whenever that doesn't cause confusion. *Users* with UID=0 are given *superuser* privileges, which allows them to act as any other *user* on the system. *Users* with UIDs less than 100 are, by convention, system *users*, while higher UID numbers are usually reserved for normal *users*.

*Users* are placed in *groups*, identified by *group* identities (GIDs). Each GID is associated with an integer in the range from 0 to a system specified maximum. *Groups* with GID=0 are reserved for system *groups*. In most installations, other *groups* with GID less than 100 are used for system functions, while *groups* with GID over 100 are used for normal *groups* of *users*.

In order to be recognized by the system as being a given *user*, it is necessary to either *login* to the system by identifying your UID, or change from your current UID to another UID. In order to prevent mistaken or malicious acts toward *users*, UNIX$^{tm}$ provides an authentication mechanism whereby *users* must authenticate their identity by providing a password to the system. UNIX$^{tm}$ transforms all passwords into an encrypted form in order to prevent attackers from exploiting copies used for comparison purposes.

Once a *user* is properly identified and authenticated, UNIX$^{tm}$ grants the *user access* all authorized information. More details of the authorization mechanism is described in a later part of this document.

Each *user* is associated with one or more *groups*. When a *user* is identified to the system, a default GID is provided and the system grants *access* to all authorized information for that *group*. A *user* can change *groups* by making an operating system request. UNIX$^{tm}$ then checks its *files* to determine whether or not the *user* is an authorized member of that *group*, and grants all authorized *access* if appropriate.

## 2.1 User and Group Control Calls

- The ACCESS call determines what *access* the current *user* and *group* have to a specified *file*.
- The ACCT call enables or disables accounting for *user* actions.
- The CHOWN call changes the *owner* and/or *group* of *files*.
- The GETUID call returns the UID and GID of the current process.
- The SETUID call changes the UID and GID of the current process.
- The STAT call gets the UID, GID, and other information about opened *files*.

## 2.2 User and Group Control Tools

- The LOGIN program allows *users* to specify a UID and password, and if they match, initializes the *user* by setting the UID and GID, placing the *user* in a system specified directory, and running a *user* specific program.
- The SU program allows *users* to change UIDs by specifying a new UID and password. If they match, the system changes the effective UID, and creates a new *process* by running the same *user* specific program that would be run at *login* for the same *user*.
- The NEWGRP program allows *users* to change from their current *user group* to other *user groups* that they are members of. If the *user* is a member of the specified *group*, the call changes the effective GID, otherwise it fails.
- The CHOWN program changes the *owner* of a specified *file*. This call only works if the effective UID is that of the *superuser*.
- The CHGRP program changes the *group* of a specified *file*. This call only works if the effective UID is that of the *superuser*.
- The WHO program lists the *users* currently logged into the system.
- The WRITE program sends messages to another logged in *user*.
- The WALL program sends messages to all *users* logged into the system.

# Chapter 3

# Processes (time)

A *process* is a single sequence of instructions being executed in an order defined by the input, output, and state of the *process* storage areas. In Unix$^{tm}$, *processes* are generated in a tree structure by a *parent process fork* ing a *child process*. The *parent process* is essentially duplicated in the *child process*, and the *child process* can then replace its memory image with the image of a desired program. Each *process* has a *process* identifier (Pid) used by the system and other *processes* to identify it. The *processes* with Pid=0 and Pid=1 are created at system bootup, and the *process* with Pid=1 is the ancestor of all other *processes* in the system.

Each *process* has a *process* identifier created by the operating system at the creation of the *process*. This is normally an integer in the range from 1 to a system defined maximum. As old *processes* "die", their Pids become available to newly generated *processes*. Each *process* also has a *process group* that may be shared with other *processes* to allow them to be manipulated together, and a *parent process* that created it. If a *parent process* dies, all of its *child processes* are normally terminated as well, but in some cases, a *parent process* dies without the *child process* dying. In such cases, the orphan *process* is adopted by the *process* with Pid=1.

In an environment with multiple *processes*, there are almost always *processes* awaiting execution. In order to control the order of execution and assure to as high a degree as possible that each *process* gets a fair chance to make progress, the system has a "scheduler" *process* that determines which *process* to grant execution time to at each "time slice". In Unix$^{tm}$, the scheduler *process* is normally the *process* with Pid=0.

Rather than encode every system service into the operating system itself, Unix$^{tm}$ uses system generated *processes* to perform processing associated with system services. This has the advantage of keeping the size of the operating system relatively small, while making it extensible to meet the needs of many environments. For example, the program that controls a printer is a *process* that only becomes ready to execute when a *file* is being sent to a printer. For two printers, you simply use two printer *processes*. For different sorts of printers, you change the parameters sent to the appropriate printer *process*.

*Processes* can communicate between each other through the use of inter*process* communications facilities, shared memory, inter*process* *files* called "pipes", signals, or regular *files*. *Processes* can be created, deleted, or modified by other *processes*, and a number of facilities exist for performing these operations.

## 3.1 Process Control Calls

- The Exit call terminates the current *process*.
- The Fork call copies the current *process*. It returns 0 to the *child process*, and the Pid of the *child process* to the *parent process*.
- The Exec call replaces the current *process* by loading an executable program into the *process* memory and initializing that program.
- The Alarm call schedules an interrupt for the current *process* as a function of system events.
- The Kill call sends a "signal" to a *process*, which the *process* can intercept and handle on its own, or in the absence of an intercept routine, terminates the signalled *process*. The "kill" signal (code number 9), cannot be intercepted, and always kills a *process* if the sender has the authority to send signals to that *process*.
- The Wait call causes the current *process* to become inactive until a sub*process* terminates. This prevents unnecessary looping while waiting for sub*processes* to terminate.
- The Signal call associates a function with the receipt of a signal. This allows most signals to be caught and handled by the *process*.
- The Pause call causes the current *process* to be suspended until it receives a signal. This saves looping while a *process* awaits an event from the environment.
- The Getpid call returns the Pid of the current *process*.
- The Getuid call returns the Uid of the current *process*.
- The Getgid call returns the Gid of the current *process*.
- The Setuid call changes the Uid of the current *process* (for the *superuser*).
- The Setgid call changes the Uid of the current *process* (for the *superuser*).
- The Umask call gets and/or sets the default protection used for *file* creation.

## 3.2 Process Control Tools

- The LOGIN program identifies and authenticates a *user*, and creates a *process* for that *user* by running a system specified program. The terminal being used by the *user* for *login* is normally attached to that *process* for input and output, and the default program run at *login* is normally the SH command interpreter.

- The SH program is designed as an interactive interface between the *user* and the operating system (commonly called a "command interpreter"). This program allows the user to specify programs, "command line" arguments, inputs to those programs, and output from those programs. It then takes the specifications and translates them into appropriate sequences of system calls. SH also has a substantial programming capability and can interpret commands from *files*, thus making is a powerful language for performing many of the tasks that would require programs in other operating environments.

- The CSH program is a version of the SH program that provides a command syntax similar to the "C" language.

- The PS program produces a list of *processes* on the system and their status. Depending on the command line parameters passed to PS, it can list *processes* from the current *user* or all *users*, and list various subsets of the available information on those *processes*. The available information includes the PID, GID, *parent*, system time, user time, I/O time, name of the calling program and command line parameters, and *process* scheduler status.

- The NEWGRP program changes the *group* of the current *process*.

- The KILL program sends a specified signal to a specified *process* or set of *processes*. It is most commonly used to terminate a *process*, but is occasionally used to send other signals.

- The AT program schedules a *process* for a later time and date. This is most often used to perform "batch" processing, schedule processing for the evening hours, or (on occasion) to do something at a particular date and time in the future. The CRON program facility is normally used for periodic tasks.

- The CRON program is a program that chronically wakes up to run programs scheduled for particular times and dates. CRON can be used to schedule specific events, but more often is used for periodic processing like remote mail delivery and reminding the operator to do backups.

- The NICE program requests that the system scheduler change the priority with which the present and all sub*processes* be scheduled to run. For normal *users*, NICE can only lower the execution priority, but for the *superuser*, NICE can be used to schedule higher priorities as well. Priorities on most UNIX$^{tm}$ systems range from -127 to 128, with -127 being the "highest" priority. *User* programs normally run at priority 0, while device drivers tend to run at negative priorities, with DMA priorities more negative then sequential devices.

- The TIME program times a sub*process*, returning the *User* CPU time, system CPU time, and real time.

# Chapter 4

# Files (space)

*Files* come in four types (Regular, Character Special, Block Special, and Directory) and are used to model storage and peripheral devices. The *file* types are used as follows:

- Regular *files* are the normal sorts of storage *files* we tend to think of when we think about computers. They are accessible either sequentially or randomly; they can be created, destroyed, and changed; they store what you put in them and retrieve it at your request.
- Character Special *files* are logical units used to represent sequential device control programs. They are sequential in that you send them sequences of symbols that are processed into output as they arrive, and they return sequences of symbols arriving from their inputs. They are device control programs in that, even though you treat them as *files* from an input and output standpoint, they actually act as programs that transform input and output requests into actions. They are character *files* in that they handle information a character at a time.
- Block Special *files* are like character special *files* except that they handle information in blocks of bytes instead of single bytes. This makes them particularly suitable for DMA devices.
- Directory *files* are actually just like regular *files* except that they store lists of *file* names and their inode numbers. An inode number is an integer associated with a system structure that holds system information about a *file*, and is designated by the system upon *file* creation.

A few examples may help to understand how these types of *files* are used. Memory, for example, is modeled by several Character Special *files* which allow *user* memory, kernel memory, or physical memory to be accessed. Physical disks are represented by Block Special *files*, while the *files* stored on those physical disks are represented by Regular *files*, and grouped under Directory *files*. Most sequential peripherals, like terminals, tape drives, and parallel ports, are represented as Character Special *files*. DMA devices like network interfaces and external disks, are typically represented as Block Special *files*.

The DIRECTORY *files* are normally used to form a tree structured logical *file*-structure, with the "/" (root) directory as the root of the *file* tree.

## 4.1 File Related System Calls

- The ACCESS call determines the accessibility of a *file* to the current *user*.
- The CHMOD call changes the protection bits associated with a *file*.
- The CHOWN call changes the *owner* of a *file*.
- The CHGRP call changes the *group* of a *file*.
- The LINK call links a *file* name to an existing *file*.
- The MOUNT call mounts a *file*-system making it accessible as a directory.
- The UMOUNT call unmounts a *file*-system from a directory.
- The PIPE call creates a "pipe" for simulating a temporary *file* used as the output of one *process* and the input of another *process*.
- The STAT call determines the status of an opened *file*.
- The UMASK call gets or sets the mask used to set *file* protections when a *file* is created.
- The UNLINK call removes a link between a *file*name and an INODE, and if there are 0 links left to the INODE, returns the *file* space and INODE to the system, thus affecting a *file* deletion.

## 4.2 File Control Tools

- The LS program lists *file* names.
- The CHMOD program changes the protection bits associated with *files*.
- The CHOWN program changes the *owner* of a *file* if the *user* running the program is authorized to make the change. Normally, this program can only be used by the *superuser*.
- The CHGRP program changes the *group* associated with a *file* if the *user* has the authority to *access* both the previous and subsequent *groups*. Normally, this program can only be used by the *superuser*.

- The Mv program changes the name of a *file*.
- The Cp program copies a *file*.
- The Ln program links a new name to an existing *file*.
- The Rm program removes a link from a *file*, and if no links remain, results in the removal of the *file* from the *file*-system.
- The Tar program converts a list of *files* into a single binary *file* suitable for placing on a tape, or unpacks the binary *file* into the constituent *files*. The Tar command also stores information about the *file*, its *owner*, its *group*, and its protection bits. Provided Tar has sufficient privileges, it can restore these values when restoring *files*.
- The Mount program mounts a *file*-structure on a directory.
- The Umount program unmounts a *file*-structure.
- The Df program lists the amount of space used, space available, and INODEs available on each mounted *file*-structure.
- The Mknod program creates an INODE.
- The Mkfs program creates a *file*-structure on a logical partition of a physical device.

## 4.3 System Files

By convention, system *files* under Unix$^{tm}$ are stored under specific names in specific places in the *file*-structure. There are a lot of *files* and directories used for Unix$^{tm}$ system *files*, and we will only describe those which are most important to systems administration and protection. Some *file*names depend on the specific Unix$^{tm}$ version you are using, but most of them are uniform across all implementations.

The root directory normally contains the following:

- The '/etc' directory is used to store highly volatile state information including the list of *users*, *groups*, passwords, terminal connections, operating system parameters, system startup sequences, the message of the day, and other such things.
- The '/bin' directory is used to store system critical executable programs, including most of the tools described in this manual.
- The '/lib' directory is used to store system libraries.

- The '/dev' directory is used to store the BLOCK SPECIAL and CHARACTER SPECIAL *files* that represent most of the peripheral devices and other physical system resources.
- The '/unix' *file* is the executable code for the operating system kernel.
- The '/usr' directory is usually used to store the 'home' directories associated with each *user*'s storage area. It is also used to store other non-system *files*.
- The '/tmp' directory is a temporary area used for storing intermediate results.
- The 'lost+found' directory in each *file*-structure is used to store lists of disk areas which are not useble for storage. This is normally because of hardware failures, but can occasionally be caused by software failures.

The '/usr' and '/etc' directories are particularly important to the systems administrator because they contain many of the critical configuration and *user files*. We begin with the *files* in /etc:

- '/etc/passwd' contains a list of the UIDs of all authorized *users*. It lists the UID, the encrypted password associated with that UID, the integer associated with the UID, the default *group* for that UID upon *login*, the name of the *user*, the directory the *user* is placed in upon *login*, and the program run upon *login* by that *user*.
- '/etc/groups' contains a list of GIDs, the integer associated with them, and the UIDs comprising each *group*.
- '/etc/TIMEZONE' stores the timezone of the computer relative to Grenich Mean Time, and is used to perform time conversions.
- '/etc/bupsched' lists the times and days at which to notify the console that it is time to perform a backup. Each *file*-system is separately listed in the backup schedule.
- '/etc/fstab' stores a list of the *file*-systems and how they are to mounted at system startup. This automates one of the many startup functions that would otherwise have to be done manually by the systems administrator at startup.
- '/etc/gettydefs' specifies a set of macros for the treatment of terminals by the *login process*. It normally specifies sequences of baud rates, control parameters, and next entries in the table.
- '/etc/inittab' specifies how each non-mountable peripheral device is to be handled in each machine state. Machine states usually include a powerfail state, a power-up self-test state, a microcode state, a single-user state, and a multi-user state. Some machines also have other states determined by the system designers. Inittab is normally used to turn devices on or off, specify how they are handled by '/etc/gettydefs', and initialize the terminal before *login*. Inittab can also be used to run a non-standard program on a terminal. For example, it could be used to automatically implement a systems administration capability, a printer controller, a batch processing mechanism, or a limited function menu system.

- '/etc/motd' contains a "message of the day" that is printed on each terminal at *login*.
- '/etc/profile' contains an SH script that is run at *login* before granting the *user process* control over the terminal.
- '/etc/stdprofile' contains a standard *login* profile copied into each *user*'s directory when they are first added as *users*.
- '/etc/termcap' contains a list of the different types of terminals and how they are interfaced with by terminal control programs. This allows thousands of different types of terminals to be handled uniformly by all UNIX$^{tm}$ programs.

The directories in '/usr' are less critical to system operation:

- The '/usr/admin' directory stores systems administration menus.
- The '/usr/bin' directory stores commonly used binary executable programs not required for systems operation.
- The '/usr/local' directory is used to store local versions of programs, and typically has 'bin', 'src', 'lib', and 'include' directories.
- The '/usr/include' directory contains header *files* included by *user* programs. These *files* store configuration or system dependent structures.
- The '/usr/lib' directory stores library *files* with commonly used subroutines.
- The '/usr/spool' directory contains spooling areas for printer queues and other spooled input and output.
- The '/usr/tmp' directory is used for temporary storage of non-system temporary *files*.

# Chapter 5

# Millions of Protection Bits, and Not a Tool to Manage Them

In a typical UNIX$^{tm}$ system, there are from 10,000 to 10,000,000 *files*, hundreds of simultaneously active *processes*, and hundreds of *users*. Each *file* and *process* has about 20 protection bits associated with it to allow the system to determine the accessibility of information. That means that it is not unusual to have millions of bits of protection information that have to be controlled in order for protection to be effective. In addition to the explicit protection information, there are many "rights" that are granted both directly and indirectly as a result of these protection bits and the available system operations.

## 5.1 Process Protection

Each *process* has its own "virtual machine" with corresponding memory, registers, input and output *files*, peripheral devices, etc. In order for *processes* to behave properly, the operating system separates *processes* from each other so that they don't interact. The cases where they do interact are controlled by the operating system.

The separation mechanism under UNIX$^{tm}$ allows a *parent process* to control a *child process* in a few ways. *Processes* can be killed, delayed, stopped, or restarted, by their *parent process*.

In addition to ancestry, each *process* is "owned" by a *user*, and a *user* can send signals to any owned *process* to affect the same general class of operations as a *parent process*.

The *superuser*, by definition, is an *owner* of all *processes*, and can thus send signals to *processes* in place of the *user* who owns a *process*.

Certain *group* operations are allowed in order to facilitate controlling large numbers of *processes*. *Process group* IDs are inherited from *parent*s, and can be changed by requesting a new *process group* from the operating system. A typical operation is to create a *process group* and run a set of interrelated *processes* from that same *group*. If an error occurs that calls for stopping the entire operation, the whole *group* can be killed without having to keep track of each *process*.

## 5.2 File Protection

*File* protection in Unix$^{tm}$ is maintained by the operating system in the INODE associated with it. Each *file* has 11 protection bits associated with it. There are 9 bits formed by the cross product of the *read*, *write*, and *execute* rights with the *owner*, *group*, and *world* sets of *users*, and two special bits that allow a program to grant its *user* the rights of its *owner* or *group*. They work as follows:

- The *read* right grants *access* to look at the contents of a REGULAR *file*, get input from a SPECIAL *file*, or examine the contents of a DIRECTORY *file*.
- The *write* right grants *access* to modify the contents of a REGULAR *file*, send output to a SPECIAL *file*, or modify the contents of a DIRECTORY *file*.
- The *execute* right grants *access* to load a *file*'s contents into memory and begin interpreting it as program. *Execute* permission only makes sense for REGULAR *files*, but in many systems is used to allow DIRECTORY search.

Each of these rights can be extended to the *owner* of a *file*, a *user* whose Gid is the same as the *group* specified for the *file*, or all *users* on the system. In most Unix$^{tm}$ systems accessibility requires recursive accessibility of all directories in the *path*, but there are some exceptions.

The Setuid and Setgid bits specify to the operating system that the respective *access* rights of the *owner* or *group* associated with the *file* are to be granted to any *process* executing the program, for the period of that program's execution.

# Chapter 6

# Typical Unix System Parameters

Most Unix$^{tm}$ systems have modifiable operating system parameters that allow the systems administrator to control performance to meet usage requirements. We will now list some typical parameters (taken from System 5 Unix$^{tm}$), and describe how they might affect performance.

- MAXSLICE is the maximum timeslice for a *process* in clock 'ticks'. Larger timeslices make *processes* run longer, so that more processing gets done between time consuming *process* swaps. Smaller time slices assure that each *process* gets activated more often, so it improves response time if each *process* has a very small task to do relatively often.
- NINODE is the maximum number of inodes that can be opened at one time on the system. Each inode is stored in a fixed location in the operating system memory area, so the more inodes, the less space is available for *user process* space. On the other hand, with too few available inodes, *file* opens will fail. Normally, we allow the smallest number of inodes we can get by with while keeping the number of failures sufficiently low for reliability requirements.
- NMOUNT is the maximum number of mounted *file*-systems at any one time. Each *file*-system requires Kernel memory space, and just like the case for *files*, we don't want to have too few for the system configuration. Since the number of mounted *file*-systems rarely changes, this can be tuned relatively tightly.
- NPROC is the maximum number of *processes* that can be operating on the system at one time. If this number is too low, *users* will keep being interrupted, and failures will be commonplace. If it is too high, the space available for *users* will cause slowed performance.
- MAXUP is the maximum number of *processes* that a single Uid can have on the system at one time. Since Unix$^{tm}$ *users* can *login* on multiple terminals at the same time, this number should be high enough to allow normal processing with typical usage. If this number is too high, a single *user* could dominate the *process* table, but if it too low, normal usage may become overly restrictive.

- FLCKREC is the size of a 'record' used for the purposes of *file* region locking. UNIX$^{tm}$ allows different *processes* to lock different sections of a *file* simultaneously, thus permitting very good performance for large databases accessed at random points. If this number is too small, the operating system has to work much harder to provide this protection, while if it is too large, the likelihood of database accesses being delayed due to a *file* lock increases.

- NAUTOUP is the delay for cached writes. If the same areas are being written repeatedly, larger delays save unnecessary writes (writes that would be overwritten again very soon), while if this number gets too large, writes are delayed so much that large portions of disk areas are still in the cache if and when the system fails.

- NOFILES is the maximum number of *files* that can be opened by a single *process* at one time. UNIX$^{tm}$ normally has many *processes* per *user*, and thus a *process* rarely has very many open *files*.

- ULIMIT is the maximum *file* size permitted by the operating system. It is not unusual for a runaway *process* to create *files* with unbounded length. If this constant is too high, these *files* can become enormous, while if this constant is too low, databases, and other programs requiring large *files*, will be limited in their utility. Large *file* sizes are particularly useful for hashing algorithms, because UNIX$^{tm}$ systems tend to have good facilities for sparse *files*, and hash *files* tend to be sparse.

There are usually about 50 tunable parameters on a UNIX$^{tm}$ system, and we have only touched on a few of them here. We are not trying to be comprehensive, and parameters differ from system to system so these won't necessarily apply to your system, but the concepts are the same. Each parameter impacts performance in a way that depends on the available physical resources and the system usage patterns. When combined, these parameters form a complex tradeoff space, but there are some guiding principles that will help you along the way.

- You almost always end up trading time with space when you tune parameters. It will be helpful to think in terms of which is the rarest resource in your environment.

- Change as few parameters at a time as possible. This lets you observe the effects of tuning each parameter.

- Don't make dramatic changes. They tend to create unuseble systems, and they may avoid optimal performance points rather than help you reach them.

- Use hill climbing to work towards optimality. Hill climbing assumes that parameters are independent, which they are not, but tends to work well in practice.

- Read the manual before making a change. UNIX$^{tm}$ manuals tell you a lot about the effects of parameter changes, which parameters depend on which other parameters, and how to recover from unusable configurations.

- Be sure to test new configurations under a range of expected loading conditions to assure that there are no dramatic effects on performance under special load conditions.

Most UNIX$^{tm}$ systems provide performance analysis tools to help understand what is dictating system performance. For example, performance tools will tell you what percentage of the system time is spent in paging or swapping, how many *file* open failures occur as a result of INODE limits, and other information that will help you find bottlenecks. Once a bottleneck has been identified, it should be pretty straight forward to find parameters related to the bottleneck and do simple experiments to find techniques for improvement.

Ultimately, the physical time and space available may simply be inadequate for your needs. In this case, you may have to add physical resources to improve performance. For example, if the number of paging faults is enormous and all related parameters have been adjusted, and the software is at its best configuration to minimize paging, you may need to add memory. If this is not possible, you may have to purchase another computer or change your usage characteristics to get the desired performance.

# Chapter 7

# Common Unix Attacks and Defenses

Although designing secure systems normally requires that we take protection into consideration through the entire design *process*, systems administrators must deal with the security issues of the system they are using, regardless of the design and implementation flaws that come with the system. For that reason, we will now list many of the most commonly exploited pitfalls of UNIX$^{tm}$ and discuss how to defend against them.

## 7.1 Trojan Horses and Viruses

**PROBLEM:** Trojan horses in the search *path* are very easy to implement in UNIX$^{tm}$, primarily because of the search mechanism used to find an executable program. Each directory in the *path* is examined for each program called. If a Trojan horse appears earlier in the *path* than the desired program, the Trojan horse is run instead of the legitimate program. A programmer can easily design the Trojan horse so that the legitimate program is executed after the Trojan horse, and thus hide the new function from the unsuspecting *user*.

PREVENTION: Trojan horses can be prevented by sound change control (a system which controls the programs entering the environment), but this doesn't work well under UNIX$^{tm}$ because *users* tend to make changes and use programs belonging to other *users*.

DETECTION: Trojan horses placed in the *path* can be detected by integrity shells, which can automatically and transparently prevent running this sort of Trojan horse.

CURE: Once a Trojan horse in the *path* is found, it is usually easy to remove by deleting the offending program and (if appropriate) replacing it with a clean copy. You should also try to find the *user* who placed the Trojan horse (perhaps by looking at the *owner* of the *file*). Be careful, it is easy to use a Trojan horse to forge another *user*'s UID on a second Trojan horse, and thus misdirect the defender.

**PROBLEM:** Trojan horses can also be used to spoof terminal *login* sessions. The attacker need only simulate the normal *login* sequence (which can be done in a few lines in SH) and save a copy of the UID and password in a *file*.

PREVENTION: Spoofing like this can be prevented by providing a 'secure' path between the *user* and the operating system, but no such facility is normally provided with UNIX$^{tm}$.

DETECTION: To detect spoofing, you can look for terminals without *users* on them which have *processes* active. Another common detection method is to look for *processes* on terminals that haven't had any I/O for a long period of time (although this is inappropriate in some environments.

CURE: Eliminating a spoofing program of this sort involves killing the *process* currently spoofing a *login*, identifying the perpetrator, and acting appropriately to prevent further attacks.

**PROBLEM:** Viruses can be used to spread an attack throughout a system or network. A virus works by replicating inside programs. Each 'infected' program then spreads the virus further. The UNIX$^{tm}$ protection mechanisms are inadequate for virus defense.

PREVENTION: Viruses cannot be completely prevented under UNIX$^{tm}$ or any other modern operating system except by eliminating sharing, or eliminating programming. This is almost never feasible in a modern UNIX$^{tm}$ system.

DETECTION: Viruses can reliably be detected by using an integrity shell instead of the normal UNIX$^{tm}$ shell. Integrity shells for UNIX$^{tm}$ have been in use for several years, and work transparently to the normal *user*.

CURE: Viruses are best cured with on-line backups which automate the restoration of corrupted information under an integrity shell. Off-line backups are also effective in many cases, as long as good detection is in place. Without good detection, backups are ineffective against viruses.

**PROBLEM:** Trojan horses can often be placed in libraries or other commonly used areas. By placing a Trojan horse in a library, it gets incorporated with legitimate programs compiled by other *users*.

PREVENTION: Publicly accessible common use libraries should not be permitted in most computing environments. A better approach is to use sound change control and have a library administrator who is in charge of examining source code and compiling all information placed in libraries.

DETECTION: There is no sound way to detect a Trojan horse in a library *file* except by examining every instruction in the *file*.

CURE: Once detected, libraries can usually be cleaned by replacing all of the corrupt *files* with legitimate copies. Without good detection, cure is infeasible.

## 7.2 Protection Bit Settings

**PROBLEM:** In UNIX$^{tm}$, there are so many protection bits, that it is inevitable that many of them will be set to potentially hazardous values. The most common problem comes when a *file* is created and the operating system assigns initial protection bits. These bits are masked by the current UMASK, which can be set by any program you run, and is often defaulted so as to allow other *users* to be granted inappropriate *access*. The threat from a wrong protection bit comes in a number of ways.

If *read* is granted when it should not be, it can be used to leak private information. If *read* is not granted when it should be, it can cause denial of services because inaccessible information may be required in order to perform a service.

If *WRITE* is granted when it should not be, it allows another *user* to arbitrarily change *files*. This makes introducing a Trojan horse or corrupting data very simple. If *write* is not granted when it should be, it can cause denial of service or lost data in cases where output cannot be stored.

If *execute* is not granted when it should be, it causes denial of services, while an *execute* permission that should not be granted may permit arbitrary *access*.

PREVENTION: The most conservative way to manage protection is to default to *owner access* only, and only grant other privileges when there is a specific need. This unfortunately requires that *users* think about protection issues if they ever want to share information. The default can be set by changing the UMASK entry in '/etc/profile' to change the default provided to *users*.

DETECTION: A common practice is to write an SH script that searches the system for protection settings that don't meet the system policy, and automatically run that script periodically. This can be used to provide a limited form of detection in simple instances, but is not a general purpose solution.

CURE: The cure to this problem under UNIX$^{tm}$ is to provide proper defaults, usable tools, and adequate *user* education about UNIX$^{tm}$ protection.

## 7.3 SetUid and SetGrp Problems

The SETUID and SETGRP protection bits are particularly useful for avoiding other protection problems, and particularly dangerous because they grant privileges that can potentially be abused. In many cases, the only way to allow multiple *users* to *access* a common area of storage without exposing them to leakage, corruption, or denial of services attacks, is to have a program that can mediate *access* on a case by case basis. This is where a SETUID program comes into play. For example, the "mail" directory is owned by the "mail" *user*, and the "mail" program is a SETUID program owned by "mail". This allows the "mail" program to control *access* to mail belonging to other *users*.

**PROBLEM:** The problem with SETUID programs is that they are granted all rights of the *owner* or *group*, and pass these rights on to any sub*processes* they create. For example, suppose the "mail" program FORKs another *process* to invoke a program called "deliver". If the "mail" program is not carefully written, an attacker could create an executable program called "deliver", put it in the beginning of the search *path*, and cause the Trojan horse "deliver" program to be executed with the rights of the "mail" *user*.

PREVENTION: We can prevent SETUID programs from being created by limiting the capabilities of the CHMOD system call. Most installations are unwilling to modify the system in this way.

DETECTION: It is common to have an SH script that is executed every night to search for SETUID and SETGID programs. They are listed for the systems administrator who is supposed to advise or control *users* who use this facility.

CURE: The cure is a two edged sword. We can simply remove the SETUID and SETGID bits from all *files*, but then we won't have the system services we might desire. Another technique is to provide a single well controlled SETUID facility which the systems administrator manages to provide any facility desired by *users*.

**PROBLEM:** Another variation on this theme is a Trojan horse that creates a SETUID program belonging to any *user* that runs it. Since the *user* running the Trojan horse has the right to create a SETUID program, and a SETUID program can extend that *users* rights to the attacker, any time you run any program, it has the potential of granting any other *user* all of your rights.

PREVENTION: There is no truly effective prevention of this attack.

DETECTION: The same detection techniques can be used as stated above. We can also augment the integrity shell used to defend against viruses to look for SETUID programs and eliminate the SETUID priviledge unless they are authorized, but this is a poor technique in this particular case.

CURE: If we have a list of authorized SETUID programs, we can automatically remove the *access* bits from *files* not authorized to have this capability, but this is rarely a practical defense against a serious attacker.

**PROBLEM:** This is particularly serious for the *superuser*. If the *superuser* ever runs an external program, the entire system could be completely and eternally compromised by that action. For this reason, it is critical for the *superuser* to be very careful about running programs or allowing the *path* to be set improperly.

PREVENTION: The *superuser* can and probably should have a more restrictive form of command interpretation than the average *user*. An example is a special form of SH that refuses to allow the *superuser* to run any program not owned by the *superuser*.

DETECTION: Detection is the same as in the above cases, except that the *superuser* is far easier to control because there are relatively few legitimate changes, and because the systems administrator can legitimately make all of the appropriate decisions without interfering with *user* rights.

CURE: Cure is simply a matter of resetting the appropriate bits. Tracking down the source of the attack is far more difficult, if not impossible. We may look at the list of all programs executed since the protection bits were last known to be correct and try to use this to track down candidates for the attack (via the 'lastlog' program). This is somewhat complex and requires a number of attacks in sequence in order to be effective, but can be done if you are patient and knowledgeable enough.

**PROBLEM:** Deamon *processes* are often used to provide shared *access* to system facilities. The paging deamon, mail handler, printer spooler, and alarm system are typical examples of facilities sometimes handled by deamons. Deamons are susceptible to all of the difficulties we have described, but in addition, they are generally operated with special privileges in order to provide their service while maintaining protection. As a result, a failure in a deamon can often be exploited to avoid protection.

PREVENTION: You can prevent demons from causing problems by only using them when necessary, and designing them well so that these problems occur very infrequently.

DETECTION: There is no systematic method for detecting security problems caused by deamons.

CURE: Without detection, cure is rarely affected.

## 7.4 Implied Protections

**PROBLEM:** In addition to the obvious protections granted by the protection bits, there are normally a very large number of rights implied by the protection state, even though they are not explicitly shown by it. As a simple example, suppose you don't have *read access* to a *file* 'X', but another *user* with *read access* accidentally leaves a copy of 'X' in a *file* 'Y' that you have *read access* to. Even though the protection seems to indicate that you can't examine the information in 'X', you can do so by examining the copy in 'Y'. This is also how a virus spreads to areas that its creator cannot directly *access*.

PREVENTION: Implied protections cannot be effectively controlled in a normal UNIX$^{tm}$ system. The only effective controls are provided by a mandatory *access* control policy based on a POset structure on information flow.

DETECTION: Implied protections can be derived, but normally, the implied protection shows that all *users* can *access* all information.

CURE: The only real cure is a system designed with POset based protection built-in. No current UNIX$^{tm}$ systems provide this.

## 7.5 Backups and Restores

**PROBLEM:** Backups may take many tapes or disks, and the programs provided with UNIX$^{tm}$ are not tolerant of any *user* errors. For example, if a tape is write locked, the backup typically fails rather than asking you to write enable the tape. On restoration, a bad tape or out of sequence tape will cause the entire restoration to fail, and it has to be started from scratch.

PREVENTION: Be very systematic and careful with UNIX$^{tm}$ backups, and use the "verify" mode if one exists on your system to assure that the backups are accurate. Clean your tape drives and test your tapes regularly.

DETECTION: The 'verify' mode detects bad backups in many cases, but the best detection mechanism is to restore tapes periodically on another system and check to assure the integrity of the restored information with cryptographic checksumming techniques.

CURE: The cure to physical problems is normally hardware maintenance.

**PROBLEM:** Restoring *files* replaces on-line versions of those *files*. If several *files* are lost or destroyed, the restoration of entire directories is usually performed as a convenience, but this may overwrite newer versions of some *files*. In databases, restoring lost parts of the database may leave an inconsistent database state. UNIX$^{tm}$ software provides options for overwriting older versions only, overwriting all version, or never overwriting, and you have to be careful to make the right choice.

PREVENTION: To prevent this sort of problem, it is common to use database integrity techniques. These can be found in good books on databases. Another good way of preventing such problems is to always deal with the database as a whole, and keep sequential backups for replay if restoration is needed.

DETECTION: This sort of problem is usually detected when massive failures result or wrong results are detected by people.

CURE: Regeneration of the database from transaction records is a standard recovery method.

**PROBLEM:** Restoring from backups typically restores *file* protections, but protection may have been changed since the backup was performed. This means that protection state changes must be redone after restoration in order to keep a consistent state. Whenever you restore or load software from a backup tape, it may create protection problems. For example, if you load a program that was saved with the *superuser* as its *owner*, you may give the *user* a method for attaining *superuser* privileges. Similar protection problems may make the *file* inaccessible after restoration without the *superuser* taking special action. When two systems have different integers associated with UIDS, a restored *file* may be accessible by the wrong *user*.

PREVENTION: The way to prevent this is to be very careful and thoughtful about how you use tapes and the storage and retrieval facility. A common practice inside an organization is to have a single set of standard UIDS and GIDS which never change, but this does not apply for imported software.

DETECTION: It is straight forward to write a customized program to check the number and location of *files* owned by each *user*, and detect large numbers of changes in these statistics. This will detect large scale problems of this sort, but is easily avoided by a sophisticated attacker.

CURE: The cure is to change the *owner* and *group* to the appropriate settings. This is easily automated for recursive directory structures via the UNIX$^{tm}$ 'find' command in combination with the CHOWN and CHGRP commands.

## 7.6 Common Directories and Libraries

Most systems have common use areas, such as the '/tmp', '/usr/tmp', and '/usr/local' areas. These are used by programs for temporary storage, or by programmers or program installers to put programs designed for other *users* to use into a common *access* area. This is efficient because it reduces the size of the search *path*, and separates systems information from local customizations.

**PROBLEM:** The problem comes when *users* can put executable programs, libraries, and other information into the search *path* of many other *users*. This allows Trojan horses and spoofing programs to be easily introduced. It also allows *users* to modify programs placed into common areas by other *users*. For example, temporary areas are used to store intermediate results during compilation. Since these areas are accessible to any *user*, any *user* can modify another *user*'s program during compilation.

PREVENTION: There is no way to prevent this in most UNIX$^{tm}$ systems, but if we educate *users* to this problem and provide proper defaults for the *path* variable, we can prevent most such problems.

DETECTION: This problem could be detected by integrity provisions in each program accessing the common use areas, but no standard UNIX$^{tm}$ program provides this sort of mechanism at this time.

CURE: If you find such an induced problem, you can replace the corrupt information.

**PROBLEM:** Spooling areas are also susceptible to attack, depending on the protections used on those directories. Since many *users* have to be able to spool output to printers, protection is commonly misapplied.

PREVENTION: The SETUID facility can be applied to secure these areas, but you have to be careful with this facility as well.

DETECTION: There is no obvious way to detect such problems, and almost certainly no general purpose method of detection.

CURE: If you can't detect the problem it is very hard to eliminate it.

## 7.7 Special Problems with Special Files

**PROBLEM:** Some of the SPECIAL *files* in UNIX$^{tm}$ map into memory areas or DMA device areas. These are used by programs like PS which *access* system areas that hold *process* tables, and WRITE which sends messages to all of the terminals. As a side effect, these areas are accessible by other programs.

For example, a program with *access* to '/dev/kmem' can *access* kernel memory, and thus watch terminal I/O buffers for UIDS and passwords. With *write access*, you can make arbitrary changes to memory while the system is in operation.

PREVENTION: In order to reduce the likelihood of these attacks, many systems setup the PS program as a SETUID program owned by the *superuser*, so any *user* running PS can *access* these areas from that *process*. They then protect '/dev/kmem' so that only the *superuser* can *access* it, thus providing PS service while preventing this attack.

DETECTION: You can detect the ability to *read files* by examining *file* protection bits.

CURE: You can cure the problem by preventing it, but ultimately, we will continue to come across similar problems with time, and you will have to deal with them as they are discovered.

**PROBLEM:** SPECIAL *files* are device drivers, and as such, they may introduce very subtle problems. For example, suppose there is a DMA device for taking video input from a camera into the system and displaying information stored in memory on an output display. If the driver is improperly designed, we can send a DMA request to the device, specifying system areas for the input location, and overwriting the operating system.

PREVENTION: The only way to prevent this problem is to be very careful when writing device drivers. In most systems, the drivers come with the system, so there is nothing you can do for prevention except limit the *users* who can *access* device drivers.

DETECTION: There is no way to systematically detect this problem without a great deal of hardware and software expertise.

CURE: The only cure to this sort of problem is prevention.

## 7.8 Locking and Simultaneous Access

**PROBLEM:** *Files* and portions of *files* can be locked by *processes* in order to control simultaneous *access* to database records or other similar information. The problem with this feature is that a *process* that never terminates can effectively deny services by locking numerous *files*.

PREVENTION: Careful programming can often prevent *file* locking problems, but to be really safe, you must be cognizant of the fact that *processes* can fail or get into odd states in a complex operating environment, and provide redundant methods for unlocking *files*.

DETECTION: The *file* locking mechanism has a facility for detecting a *file* lock. Most programs that deal with locking problems of this sort use retries to check for extended locking conditions.

CURE: *File* locking can normally be terminated by terminating the *process* that has the lock on the *file*. In cases where deamons are controlling database *access*, the deamon may have to be restarted unless it is designed to handle this situation. Pairs of deamons can be used to form a self-test and repair facility to assure that no single *process* failure will lock a file indefinately.

## 7.9 Cleaning Files Before Use

**PROBLEM:** Space from deleted *files* is reallocated to other *files*. If the contents of reused disk areas are not cleared before reuse, the deleted information can be recovered by the next reader. Many UNIX$^{tm}$ systems automatically clear deleted *file* areas, but in some systems this does not happen, and the *users* should be provided with a secure deletion program.

PREVENTION: Most UNIX$^{tm}$ systems automatically clear the contents of *file* areas before granting *access*, but there are some exceptions.

DETECTION: The easiest way to detect this problem is by performing an experiment. Create a *file* consuming most of the free space on the system, and fill it with a known pattern (e.g. the integers from 1 to the *file* size). Delete the *file*, and then create a new *file*, seek through a series of locations that have not been written, and read them to determine if they have any of the written pattern in them.

CURE: There is no cure for the underlying operating system problem except the repair of the operating system itself, but the RM program can be modified to wipe the *file* clean upon deletion. This will make the vast majority of *user* deletions secure. This can be augmented with a periodic wiping of unused areas of the disk to improve protection against this attack still further.

## 7.10 File Size Limits and Number of Files Open

**PROBLEM:** *File* sizes are limited by the operating system to prevent runaway *files* and limit domination of resources. In systems with inadequate disk space, it is common for *users* to preallocate large amounts of space for themselves using large *files*. When system space is low, they can free space for themselves as required.

PREVENTION: The real solution to this problem is to provide sufficient space for the *users* on your system. This will make it unnecessary for them to protect themselves from denial of services.

Detection: The Df program can be run to check on the available *file* space, and the 'du' program can be used to generate a recursive listing of space utilization.

Cure: If insufficient space is available, you should consider clearing temporary storage areas and log *files*. If this doesn't yield enough space and purchasing additional space is not an option, you should consider other alternatives. A listing the space used by each *user*, sorted by most space first and posted for all *users* to see may induce social pressure for space reduction. Automated programs to look for and/or delete automatically generated *files* such as 'core', and intermediate compilation *files* can help reduce space. Replacing 'spaces' with 'tab' characters in some text *files* can save large amounts of space. A *file* compression system can be used to compress infrequently accessed *files*. These *files* can also be moved off-line in some cases. Many other options are possible.

## 7.11 File Systems Gone Bad

**PROBLEM:** A little while ago, a cleaning lady sprayed one of my disk drives with cleaning solution, and very nearly caused a disk crash. In fact, there were many transient errors, and the system reported a disk crash, but through some miracle, the system did not go down. I did an immediate backup onto tape, and all was well (whew!). The greatest fear of the computer *user* is not death by fire, it is a disk crash.

Under Unix$^{tm}$, *file* systems are not completely stored on the disk. In order to enhance performance, many systems keep portions of the *file* system in a memory cache area. As a result, if the system is simply turned off, there may be an inconsistent state stored on the disk. Fortunately, Unix$^{tm}$ *file* systems normally contain enough redundant information to recover from many such problems. The recovery *process* is normally performed as an automatic consequence of bootup disk checks, but in some cases systems administrators have to cleanup disks during normal operation.

Prevention: *File*-system failures cannot be completely prevented, but there are some important techniques to help reduce the rate of occurrence. The most common fault leading to a *file*-system failure is a power failure. Because most Unix$^{tm}$ systems cache *file*-system changes to enhance performance, a power failure at the wrong time may be catastrophic. The best defense is an uninterruptable power supply (UPS). In my facility, we experience power failures or serious fluctuations more than 20 times per year. Without the UPS, we would have massive problems, but with the UPS, we haven't lost *file* information in over 15 years of timesharing under Unix$^{tm}$.

Detection: *File* system crashes are very easily detected, because Unix$^{tm}$ systems perform automatic self-test at bootup. They also tend to produce obvious and dramatic results.

Cure: The only real cure for otherwise irreperable *file*-system failures is restoration from backups.

## 7.12 Self-Replicating Processes

**PROBLEM:** Occasionally, a *user* creates a *process* that repeatedly spawns copies of itself. Each copy also replicates, and so on. The net effect is that it becomes nearly impossible to eliminate all of these *processes* without a system reboot. The reason it is so hard to get rid of this sort of *process* is that each time we kill one copy, another one is created to take its place.

PREVENTION: We can limit the number of *processes* available to a *user*, or otherwise restrict the rate of *process* spawning to slow the effect, but ultimately, we cannot deny *users* the ability to use the capabilities of UNIX$^{tm}$ and still have a workable environment.

DETECTION: Detecting run away *processes* is usually pretty easy because the *process* table fills and this produces error messages on the console. In some cases, the *user* will let you know that they made a mistake and cannot *access* their account. Thrashing may also be an indicator of this problem.

CURE: The only way to stop a self-relicating *process* during normal operation is to place a very high priority on the *processes* killing the worm segments. This rarely succeeds, and more often than not, a system reboot is required.

## 7.13 Paging Monsters

**PROBLEM:** A paging monster is a program designed to force the operating system to slow down by forcing as many pages as possible out of memory every time it is run. A typical paging monster loops, writing one byte to each of 1,000 or more different memory pages. If we combine the paging monster with a worm, we can sometimes almost halt all other processing. As a side effect, we tend to exercise some of the less tested parts of the operating system under high stress conditions, thus bringing out latent programming or design errors that would otherwise show up very rarely. This can cause anything from a crash to a protection failure.

PREVENTION: Paging monsters cannot be prevented in and modern UNIX$^{tm}$ system.

DETECTION: Paging monsters can be detected by the characteristic system thrashing.

CURE: The cure for paging monsters is to kill the offending *processes*. If the paging monster is also self-replicating, we typically have to reboot the system.

## 7.14 Signals and Situations

**PROBLEM:** The signalling system under Unix$^{tm}$ is essentially a model of a priority interrupt handling system. Since this is an asynchronous operation that is not controllable by the program, the number of different machine states that could apply when the signal is handled is enormous. As a result, it is impossible to fully test interrupt handling, and without great care in design, it may be impossible to get it right.

A simple interrupt handler might store the registers on the stack, and call a handler routine. Upon return, the handler restores the registers from the stack, and proceeds to do whatever it was doing before. This works well, as long as the stack is not nearly full when the interrupt takes place. If there is a stack overrun, it causes another interrupt, which in turn tries to push onto the stack, and off we go into an infinite recursion.

Prevention: A theoretically sound interrupt handling system would solve the problem, but there is no practical way for a systems administrator to eliminate this problem. System designers often provide less than the worst case resource requirements because it is too expensive to cover relatively unlikely circumstances.

Detection: There is no simple way to detect this particular problem in an operating system.

Cure: The system is normally rebooted after this sort of problem because normal operation deteriorates rapidly and a system crash is often inevitable.

## 7.15 Tying Up the Console

**PROBLEM:** The Unix$^{tm}$ console is critical to system operation, and to the extent it is tied up, the system may not be able to complete critical tasks. For example, many Unix$^{tm}$ systems will stop operating if the system console is turned off.

Prevention: Only *login* to the console to perform system administrative tasks and keep the console power turned on whenever the system is in operation.

## 7.16 Limited Function Rarely Is

**PROBLEM:** Many products are designed to provide nominal protection if their function is limited. They limit their function by providing a command interpreter or other mechanism that is supposed to prevent the *user* from bypassing controls, but in many cases, this can be bypassed by a clever enough *user*. For example, many such system are invoked by SH or CSH, and can be "core dumped" by typing an appropriate keyboard combination. Once halted, they may revert to the command interpreter, leaving the *user* logged into a privileged UID. It is usually a mistake to believe vendor claims about protection matters, since most vendors are not knowledgeable enough to provide sound protection.

PREVENTION: The first step towards a solution to this problem is using a limited function program in place of SH at *login*. This is done by modifying the '/etc/passwd' file to run the limited function program instead of SH or CSH.

DETECTION: Sometimes testing reveals these flaws, but there is no reasonable amount of testing that can assure proper operation. In other words, by experimenting we can prove programs wrong but not right.

CURE: There is rarely a cure for vendor supplied software, but it is a good idea to use vendor protection only as an augmentation of system protection mechanisms instead of trusting it implicitly.

## 7.17 Serving Adequate Notice

**PROBLEM:** Many administrators are unaware of the possible legal requirement to serve notice to *users* and potential attackers of the limits associated with using a system.

CURE: Any legal notice required can be printed for each *user* at *login* by putting the notice in the '/etc/motd' *file*. The message of the day only prints a message, and you might consider a verification program that prompts to assure that the *user* has read the message. This can be implemented in '/etc/profile', which is interpreted at every *login* using SH or CSH.

## 7.18 Password Guessing

**PROBLEM:** Most *users* use easily guessed passwords, like their UID spelled backwards, their phone number, or their first or last name. This makes it very easy for an attacker to break into the system.

Prevention: On most Unix$^{tm}$ systems, the password changing program only permits passwords of a minimum length and requires that they have both letters and numbers or special characters. This too leaves many easily guessed passwords.

Detection: Password testing programs can be used to guess obvious password or tell a user that a password is too obvious prior to its use.

Cure: In most modern Unix$^{tm}$ systems, passwords must have a minimum length and be made up of characters and numbers, but far better password restrictions can be easily implemented is a source version of the 'passwd' program is available for customization. In some cases, a cryptographic time or use dependent key is appropriate for further protection, and in still fewer cases, biometric devices may be appropriate.

Prevention: Another feature of many Unix$^{tm}$ systems is password aging. In this scheme, passwords must be changed periodically in order to remain valid. This prevents guessed passwords or passwords gleaned by other means from being used for an extended period of time.

## 7.19 The Time of Day

**PROBLEM:** Many operations depend on the time of day. For example, partial backups are based on modification times, external systems may be polled at certain times of day, maintenance tasks may be time dependent, etc. A common protection technique is to add a time-lock to *user login* so that they can only *login* from a particular terminal at a particular time. By changing the time of day, all of these time dependent functions can be made to happen at the wrong time. This can also produce very subtle problems, for example, a remote system being polled at the wrong time may prevent proper sending and receiving of mail.

Prevention: Normally, the clock can only be changed by the *superuser*, so good general protection practices will prevent almost all such problems.

Detection: The 'date' command will show the wrong time.

Cure: The *superuser* can set the date and time with the 'date' command.

## 7.20 Thrashing and Performance Analysis

**PROBLEM:** Systems often perform poorly under load conditions they are not configured for. For example, if the physical memory is small and there are a lot of *processes*, there may be a lot of paging of memory to disk and back. This is called thrashing. The same system may perform very well with a slightly different load distribution. An attacker can exploit this to cause denial of services.

PREVENTION: This problem can only be prevented by providing excess resources, which costs more money than a more prudent amount.

DETECTION: Thrashing can be detected by slow system performance, and analyzed by the performance analysis tools provided with your system.

CURE: Adjustments are often required in order to keep performance high. For example, by reducing the size of the internal tables, adding more physical memory, getting a faster disk, modifying the usage patterns, or modifying key system facilities, we might dramatically change performance.

## 7.21 Network Attacks

**PROBLEM:** In a computer network, it is usually very easy to lie about who or where you are. for example, most networks use a node identification number to determine whether another node has *access* rights to information. By simply changing the node number in the configuration phase, it is usually possible to forge an identity. Once a forged machine identity is given, it is usually very simple to forge a UID. The way current peer networks are designed, once a remote machine is identified, a peer across the network may get equivalent *access* in a peer machine.

PREVENTION: This sort of attack can be prevented by taking a very conservative attitude towards networks. For example, you can prohibit most network services, or control network services very tightly. Requiring more authentication than modern networks require by default also helps a lot.

DETECTION: By using the PS facility, you can detect the presence of *processes* and how they are connected. Automated programs can be used to look for patterns that tend to indicate excessive network behavior, and system log *files* can be analyzed to detect external entries. If you have a very restricted environment, this works well, but in many environments, the usage patterns are not predictable enough to differentiate attacks from legitimate *access*.

CURE: The cure is a high degree of isolation from external systems.

**PROBLEM:** Any system connected to a network can potentially exploit packets sent across the network. Any time a remote *login* is performed, a Uid and password are sent over the network. In non-encrypted networks, these packets can be observed, and *access* to the remote node can thus be attained. The same is true for sensitive information. In fact, it is quite simple for any node in a network to modify packets passing through it (in a token ring architecture), or introduce packets into a sequence of packets used to communicate between two other nodes. Finally, it is very easy to deny services in most networks, either by creating a network virus that consumes all of the network resources, or by logically or physically overloading the network.

Prevention: The best prevention against low level attacks of this sort is a sound encryption and authentication standard in the network, and a set of well designed network protocols. Unfortunately, most modern networks have known problems in these areas, and a systems administrator cannot easily change this.

Detection: Some network attacks are easily detected, but as a rule, there is no systematic way to tell that these attacks are taking place. One technique that is used fairly widely is a program that tracks load parameters and detects changes as an indicator of problems.

Cure: In some cases, the only cure to these problems is a complete shutdown and restart of the entire network. In other cases, even this may not eliminate the problem.

**PROBLEM:** In peer networks, there are some rather complex protection related problems that make network wide control very important. For example, two well controlled nodes in a peer network may combine to create an opening. If one node ($P$) uses physical *access* controls to prevent system entry, and the other node ($L$) uses logical controls to prevent entry, a maintenance person adjusting a printer on $L$ may introduce a virus that enters $P$ over the network, and spreads back to the $L$ via the peer equivalence of a *user* in both $L$ and $P$.

Prevention: This sort of problem can only be prevented by prudent network design and implementation, with a uniform network protection method and an analytical basis for believing it will work properly.

Detection: There is no systematic detection method.

Cure: The cure is usually starting from scratch and designing the network controls properly.

**PROBLEM:** File-servers are commonly used to share information between computers in networks. Any protection problem in the server therefore has the potential of causing network wide protection problems. For example, a virus on a file-server will usually spread throughout the network very quickly. A file-server failure might cause widespread denial of *access*, or even make many of the systems in the network inoperable. In essence, file-servers make all of the nodes in a network act as part of a single larger timesharing system. This means that any problem in one system is a problem in all of the systems.

Prevention: The best way to assure that file-servers are not exploited is to concentrate protection efforts there. This is simply a matter of spending more on defense for systems with higher exposures.

**PROBLEM:** If file-servers help to spread problems in a local network, gateways extend these problems to a still larger group of systems. The protection advantage of gateways over file-servers, is that they usually limit the interaction between their subscribers, whereas a file-server acts to integrate the clients. The disadvantage is that anything that can pass the gateway can potentially affect on a much larger set of systems. For example, the 'Internet Worm' of 1988 spread through gateways, and thus affected about 60,000 systems. Another example was the attack described in 'The Coocoo's Egg', where an attacker used legitimate *access* paths and guessed passwords to leap from machine to machine across global networks.

Prevention: Just as in the case of a file-server, the increased exposure of a gateway should lead to increased expenditure on protection.

# Chapter 8

# The Jobs Systems Administrators Do

Systems administrators under Unix$^{tm}$ are required to perform a wide variety of tasks. In some of the larger installations, there may be tens or even hundreds of people involved in the administrative and support functions, but for the vast majority of Unix$^{tm}$ systems, the systems administrator is just one of the users who has used Unix$^{tm}$ more than the other users, or a programmer who became the administrator by an accident of fate.

As a rule of thumb, about 5-10% of the computing budget of most companies is spent on systems administration and security. That means one systems administrator for every 10 *users*, or almost one hour per day per *user* spent in systems administration. In large installations, systems administrators only perform a small number of tasks, while in smaller systems, the administrator may take on many roles.

## 8.1 Partially Automated Tasks

Administrators in a Unix$^{tm}$ environment clearly have a significant number of duties to perform. Some of the duties that are partially automated in most Unix$^{tm}$ systems include:

- Diagnostics are performed to find hardware faults that have caused abnormal system behavior, and at system startup to assure that the hardware is operating properly before loading the operating system. This includes tests for memory, disks, and other system critical and peripheral devices.
- Disk administration is performed to control the partitioning of disks into *file* areas, the swap space used to store *process* memory not currently executing, system overhead areas, and areas of disks that are not usable. This normally includes adding or removing disks and changing disk allocation, and in most cases destroys all information on the affected disk in the process.

- *File* administration is performed to control *files* within disk areas. This normally includes moving *files* around, deleting *files*, moving *files* to off-line backups, restoring *files* from off-line backups, and controlling the allocated area for *files* of different *users*.
- Machine administration is performed to change machine dependent parameters. It normally includes adding and removing peripheral devices, setting machine names and *login* messages, and similar activities.
- Package administration is performed to control software that is not a built-in part of the operating system. For example, windowing systems, network control programs, databases, languages, and other "application" programs are sometimes handled this way.
- Software administration is performed to control software that comes with the operating system, but is not part of the kernel. For example, printer control programs, system editors, and the other utilities that come with UNIX$^{tm}$ are normally loaded this way.
- System Setup is usually performed once to tell the system about local parameters. It consists primarily of telling the computer what time zone it is in, setting the initial system clock, setting the system name and *login* message, and changing system passwords from their defaults (anyone can use them without a password as the default).
- Tape administration is performed to deal with the tapes usually used to move *files* from system to system and backup *files* on a periodic basis in case of loss of data. Tape administration involves formatting tapes, testing them for proper formatting, performing maintenance on tape heads and drives, and resetting the usage counts used to assure that tapes don't become ineffective due to overuse.
- Terminal administration is performed to control which terminal lines are used for which purpose. For example, terminal lines may be used for printers, remote modem *access*, to connect two computers, or for *user* terminals. You can set speed and other line parameters for each terminal line, and control whether the line will prompt for *login* or be accessible from *user processes*.
- *User* administration is performed to control which people are allowed *access* to which parts of the system. It consists of adding or removing *users* and *groups*, and determining where their storage areas are kept.

## 8.2 Manual Tasks

Hardware and software specification, acquisition, installation, and maintenance are done on a case by case basis. Most products that are designed to be used by UNIX$^{tm}$ either operate on standard interfaces, or come with software designed to interface with UNIX$^{tm}$. An important part of the administrator's job is to make sure the interface to UNIX$^{tm}$ will operate.

Another major systems administration task is tuning tunable parameters. This was described in some detail earlier, and we will not elaborate further here.

Performing backups is relatively automatic in the sense that a menu system is provided to automate the typing of commands related to the backup, but people still have to do the backups and test them to assure that they are working properly. Periodic tape head cleaning is required in order to keep backups reliable. Off site copies of backups are important in case of flood, fire, or theft.

Disaster recovery programs should be in place and tested periodically to assure that operation can be restored in case of emergency. Simply having a plan is inadequate, because most sites find that the plan doesn't work unless it has been thoroughly tested.

Interfacing with networks and external systems is the source of many horror stories. UNIX$^{tm}$ administration facilities are particularly obscure for the "uucp" facility, and require laborious trial and error combined with a lot of reading, but once they are working, they only have to be periodically monitored and managed. Other networking systems are equally complex to manage.

Assuring proper delivery of mail and other system provided services takes a considerable amount of administration effort. One of the major problems comes from errors in these services that are not detected early. A good systems administrator will not wait for problems, but rather will have an ongoing set of tests that tend to find problems before the *users* become aware of them. As procedures become well known, they can often be automated to a large degree, and then only the proper operation of the automation has to be tested.

Periodic checking and clearing of audit trails is very important, since these tend to grow without bound if left unattended. Again, the these operations are often automated by the more skilled systems administrator.

Setting up terminals and other peripheral devices for *users* grows with the number of *users*. A good rule of thumb is four terminal setups or changes per year per *user*, but this varies greatly.

Setting up and maintaining system deamons, controlling the facilities they provide, and providing interfaces to those services for *users* can be a big job if the systems administrator is responsible for these services and if a lot of services are expected from the system.

Interacting with vendors and maintenance personnel is another important administrative job. In most cases, maintenance and vendor personnel will do the easiest thing they can do to get their job done, but it may not be the best thing for your system. You have to watch and manage them in order to get the best results.

Setting up and maintaining system service UIDs is another common administrative task. Setting up these UIDs is vital to system protection, but is often not done because of ignorance or laziness by the systems administrator.

Protection administration, at a minimum, consists of assuring that the attacks we discussed earlier are protected against to a degree appropriate for the environment.

# Chapter 9

# Controllable Configurations

From our previous discussions, it would seem to the casual reader, that UNIX$^{tm}$ is impossible to administer effectively, and yet there are many well managed UNIX$^{tm}$ systems. The better managed systems tend to have a few things in common that make them easy to administer effectively.

## 9.1 Structure Reduces Complexity

One of the most important steps toward effective control of a complex system is the introduction of structure. By that I mean that a completely unstructured UNIX$^{tm}$ environment; where each *user* is treated differently, each terminal is treated differently, any kinds of *files* are placed anywhere in the system, etc.; is completely unmanageable. The complexity quickly becomes too high for humans to manage. In a well structured environment, it is always easy to find what you are looking for and put things where they belong.

UNIX$^{tm}$ file systems have historically been structured according to standards created by the early *users*, and many of these standards are now so deeply embedded into the UNIX$^{tm}$ environment that they sometimes seem to be a part of the operating system itself. The most obvious set of standards come from the names of directories:

- **/** is for system critical *files*.
- **/usr** is for non-critical *files*.
- **bin** is universally used to store binary executable programs. The system areas are called '/bin' and '/usr/bin', while local versions and commonly used local binaries are usually kept in '/usr/local/bin'. Each *user* typically has a 'bin' directory in their home directory, and each package that is installed typically has 'bin' areas used to store its binary files.

- **lib** is universally used for unlinked compiled binary images incorporated into other binary executables. The '/lib' and '/usr/lib' system areas, '/usr/local/lib', and the lib areas within user file areas and package specific areas correspond to the case for 'bin' directories.
- **include** is universally used for 'header' files which include details about system specific parameters and structures that are compiled into other programs.
- **src** is universally used for source programs.

## 9.2 Using Environment Variables

As systems become larger, more standards are needed to keep track of the increasingly complex configuration. As systems evolve, there is also a tendency for large portions of *file* structures to be moved between physical areas. Less experienced *users* and administrators have to do enormous amounts of work every time such a move is made, because the embed pathnames too deeply within the system. More experienced experts eliminate this problem by using the environment variables very extensively.

At *login*, each *user* normally interprets the '/etc/profile' file. In well controlled systems, a local variant (stored perhaps in '/usr/local/etc/profile') is interpreted at *login* to create a set of environment variables which are used by programs to locate *files*. In Sh and Csh, environment variables are directly available, while programs can access these variables through the 'getenv' subroutine call. In a system with multiple applications, each application may have many associated environment variables.

There are several advantages to the use of these environment variables. One major advantage is that it can increase performance by eliminating large *path* searches, thus reducing program startup time and eliminating many potential locations for Trojan horses. Another major advantage is the ability to move applications transparently. If the software is properly implemented, no program changes will be required to change the directory a set of files are stored in. For installation, we simply put the software in the most convenient place and add one line to the system profile to specify where *files* are to be found.

## 9.3 Administrative Automation

One of the saving graces of Unix$^{tm}$ administration is the ease of automating tasks with Sh and Csh command scripts. As you perform more and more administrative tasks, you become better and better at figuring out how to make them essentially automatic. Sh provides a convenient way of automating commands, but more than that, it is a general purpose interpretive language.

Another UNIX$^{tm}$ facility that helps in administration is its 'program' maintenance facility called 'Make'. Make is a program that allows the programmer to describe the relationship of *files* to each other, and provides automation for reflecting source changes in destination files. Many *users* use 'make' to rebuild executable programs and libraries to reflect changes in a source program, but the facility is general enough to automate many other aspects of systems administration.

Between these facilities, it is pretty easy to automate the vast majority of administrative tasks, and in fact, that is one of the major tasks of a good UNIX$^{tm}$ systems administrator.