Elements of Security: AIX 4.1

Document Number GG24-4433-00

October 1994

International Technical Support Organization
Poughkeepsie Center

Take Note!

Before using this information and the products it supports, be sure to read the general information under "Special Notices" on page ix.

First Edition (October 1994)

This edition applies to the initial releases of AIX Version 4.1 for RISC System/6000.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader's feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept 541 Mail Station P099
522 South Road
Poughkeepsie, New York 12601-5400

When you send information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1994. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Abstract

This document discusses many of the security-related elements of AIX 4.1. It is directed toward a reader who is a system administrator for one or more AIX systems, although much of the material may be useful to AIX users. Recommendations and suggestions for installation and day-to-day administration are included. Specialized topics, including DCE and NIS, are not discussed. Basic UNIX knowledge is assumed.

(120 pages)

© Copyright IBM Corp. 1994

Contents

Abstract . . . iii

Special Notices . . . ix

Preface . . . xi
Acknowledgments . . . xi

Chapter 1. Introduction . . . 1
1.1 Security Policy, Standards, Guidelines . . . 1
1.2 Who Needs Security? . . . 2
1.3 How Much Security? . . . 3
1.4 System Categories . . . 3
1.5 Common Security Exposures . . . 4
1.6 Physical Security . . . 4
1.6.1 ″Power On″ Hours . . . 6
1.7 System Administrator . . . 6
1.8 Computer Security Audits and Reviews . . . 7

Chapter 2. AIX Security Structure . . . 9
2.1 smit . . . 11
2.2 Visual System Manager . . . 12

Chapter 3. User Accounts . . . 13
3.1 User Identification, UID . . . 13
3.1.1 The root User . . . 14
3.2 Single-user Workstations . . . 15
3.3 Users . . . 16
3.3.1 User Parameters in Smit . . . 17
3.3.2 System Defaults . . . 20
3.3.3 Shadow Files . . . 20
3.3.4 Passwords . . . 21
3.4 Search PATH For User . . . 24
3.4.1 Timeouts . . . 25
3.4.2 Prompts . . . 26
3.4.3 Disabling the root Userid . . . 26
3.5 Groups . . . 27
3.5.1 AIX Group Usage and Administration . . . 27
Group Usage for Workstations . . . 29
3.6 Standard Userids . . . 29
3.7 Files Associated With User Accounts . . . 30
3.7.1 Additional Authentication Methods . . . 33
3.8 Verifying the User Environment . . . 33
The grpck Command . . . 34
The usrck Command . . . 34
The pwdck Command . . . 34
The lsgroup and lsuser Commands . . . 34
The tcbck Command . . . 35
3.9 Other Topics . . . 35
3.9.1 Repairing the root Userid . . . 36
3.9.2 Password Cracker Programs . . . 37

© Copyright IBM Corp. 1994

Chapter 4. AIX File Security . . . 39
4.1 File Systems . . . 39
The mount Command . . . 41
4.1.1 Private File Systems . . . 41
4.1.2 Inodes and Links . . . 42
4.1.3 Ownership . . . 43
4.1.4 Permission Bits (Basic) . . . 44
4.2 Basic File Security Concepts . . . 45
4.2.1 The ls Command . . . 47
4.2.2 Permission Bits (Advanced) . . . 49
Directory Permissions Summary . . . 51
4.2.3 The umask Variable . . . 52
4.2.4 File Timestamps . . . 52
4.3 The ACL Commands . . . 53
Base Permissions . . . 53
Extended Permissions . . . 53
The chmod Command . . . 56
4.4 Files That Grow . . . 56
4.5 AIX Version 4 Error Logging . . . 57
4.6 Other Comments . . . 58
4.6.1 Unowned Files . . . 59
4.6.2 The /tmp Directory . . . 59

Chapter 5. Network Security . . . 61
5.1 Physical Communication Security . . . 61
5.2 Network Security Goals . . . 62
5.3 The securetcpip Command . . . 63
5.3.1 Remote Login Controls . . . 64
The /etc/hosts.equiv File . . . 64
The .rhosts Files . . . 65
The .netrc Files . . . 65
5.3.2 Other Important TCP/IP Files . . . 66
The /etc/hosts File . . . 66
The /etc/inetd.conf File . . . 66
Name Server . . . 66
5.3.3 The netstat Command . . . 67
5.4 Network File System Overview . . . 67
5.4.1 The /etc/exports File . . . 68
5.4.2 NFS Support for ACLs (Access Control Lists) . . . 69
5.4.3 Secure NFS Operations . . . 69
5.4.4 The Client - Server DES Interaction . . . 71
5.5 Network Information Service (NIS) . . . 71
5.6 Adapter Security Levels . . . 73

Chapter 6. Logs and Accounting . . . 75
6.1 AIX Log Files . . . 75

Chapter 7. Trusted Computing Base . . . 77
7.1 TCB Description . . . 78
7.2 Using the tcbck Command . . . 78
7.3 Using the Trusted Login and Trusted Shell . . . 79

Chapter 8. Auditing Functions . . . 83
Audit Events . . . 83
Audit Objects . . . 84

Information Collection 84
Audit Commands 85
8.1 Audit Configuration 85
8.2 Basic Audit Usage 87
Basic BIN Auditing 87
Basic STREAM Auditing 88
Basic Object Auditing 88
Minor Comments 89
8.3 Recommendations for Auditing 89
8.3.1 Audit Limitations 89
8.3.2 Auditing Products 90

Chapter 9. Other Topics 91
9.1 Firewalls 91
9.2 X Windows 92
9.3 The skulker Script 92
9.4 Controlling cron and at 93

Chapter 10. Checklists and Reviews 95
10.1 Planning 95
10.1.1 Initial Installation 95
10.1.2 Continuing Activities 97
10.2 Reviewing a System 99

Appendix A. DoD Classes 105
A.1.1 Levels for Commercial Users 107
A.1.2 Comments 108

Appendix B. Additional Authentication 111
B.1 Two-person Login 111
B.2 Password and Local Program 111

Appendix C. Audit Events 115

Index 119

Special Notices

This publication is intended to help you to understand and implement the basic security elements of AIX Version 4.1 The information in this publication is not intended as the specification of any programming interfaces that are provided by AIX Version 4 or by any subsystem or product used with AIX. See the PUBLICATIONS Section of the IBM Programming Announcements for AIX Version 4 or for associated products for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this document is not intended to state or imply that only IBM′s program product may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program, or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial relations, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, USA.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

The following terms are trademarks of the IBM Corporation in the United States and/or other countries:

AIX	IBM
InfoExplorer	NetSP
AIXwindows	RISC System/6000

The following terms are trademarks of other companies:

UNIX	Developed and licensed by AT&T; the trademark is now controlled by X/Open.
Network File System (NFS)	Sun Microsystems, Inc.
INed	INTERACTIVE Systems Corporation.

Preface

This document provides an overview of AIX security elements and recommendations for their use. It is directed to newer AIX installations. Basic elements of TCP/IP networking security are considered. Approaches for auditing the AIX security environment are discussed. The reader is assumed to have a basic working knowledge of UNIX.

The studies were done with the first release of AIX 4.1. Later updates and releases may change some details, but the basic recommendations should remain valid.

The material in this document should be used in conjunction with the formal AIX documentation, both printed and on-line. Comments included in various AIX files, as distributed by IBM, are also relevant and these are referenced in the text.

The first chapter discusses the need for security, levels of security, and background information which may be useful. The second chapter briefly discusses the overall basis of security in a UNIX system.

The third and fourth chapters discuss user administration (security controls) and file security controls, in some depth. First elements of network security are discussed in Chapter 5. Log files created by AIX are discussed in Chapter 6.

The Trusted Computing Base (TCB) of AIX is described in Chapter 7, with recommendations for its use. AIX's auditing functions are described in Chapter 8. These TCB and auditing functions are unique to AIX and are important for security administrators. Chapter 9 contains brief reviews of other topics not covered elsewhere.

Chapter 10 contains a lists of specific installation suggestions, and a substantial list of specific checks to be used using a security review of an AIX system.

Acknowledgments

This document is the result of a project of the International Technical Support Organization in the Poughkeepsie Center. The primary author was William R. Ogden.

The following people provided assistance for this document:

- Lee Terrell - IBM AIX Development (Austin)
- Kurt Meiser - ITSS International, Inc. (Poughkeepsie)
- Jean Nance - IBM AIX Development (Austin)

© Copyright IBM Corp. 1994

Chapter 1. Introduction

In most UNIX environments, security administration is considered to be a subset of general system administration. There are rarely separate security administrators. The term *UNIX administrator* covers a very wide range of tasks, skills, and requirements.[1] A new AIX user who installs his own UNIX system (following a vendor's "quick installation" instructions for loading the system from a CD-ROM) is his own system administrator. A person with years of UNIX experience, working as a full-time administrator in a large, complex networked environment, is also a UNIX administrator. There is a wide skill gap between these two extremes.

This document is intended for a newer, less experienced AIX administrator. It covers some of the background concepts important for security, and addresses the basic elements of AIX security.

1.1 Security Policy, Standards, Guidelines

″**Security is a management function.**″ This statement is repeated in any discussion of security -- and is often ignored. No reasonable combination of hardware and software security products will be effective without a proper management environment, involving all levels of management. Every organization needs a security policy. It may be very simple, but it must be understood by everyone in the organization.

Security policies, and their implementation, have been discussed in many documents. These discussions will not be repeated here. However, they are important; it is simply wrong to implement pieces of security without a reasonably clear goal in mind. A security policy should outline (in some detail) the security goals of an organization.

Security policy considerations should include:

- The security boundaries between individuals, groups, the organization, and the rest of the world. What types of data must be restricted to specific individuals? What is shared by departments or other groups? What is available to everyone in the organization? What can be freely distributed outside the organization?
- How should the organization be divided into groups (for security purposes)? This may not always follow departmental lines. (In practice, considerable effort is needed to produce *good* group definitions.)
- What levels of security are needed for these various functions? It is important not to overstate the needs. The requirements of a clearing bank differ from those of a clothing manufacturer. This would include rough classifications based on confidentiality (controlled *read*) and integrity (controlled *write*).
- Where is the line between *nice to have* and *required* security? Specifying too much security (to cover all the *nice to have* situations) can detract from

[1] The term "administration" tends to include many functions that were termed "systems programming" on mainframe systems.

© Copyright IBM Corp. 1994

required security. The trivial statement "All our company information should be secure" is not a reasonable starting (or ending) point for a security policy.

- What is the cost of security? What is an acceptable cost? Administration and inconvenience to users are parts of the total cost. What is the *risk* and *cost* of lost or corrupted[2] or compromised data? In many cases a reasonable risk/cost analysis (using high/medium/low categories rather than monetary costs) can eliminate many of the nice-to-have items from a security-controls list.
- Who administers security? Who monitors or audits it? These functions require time and skill. No amount of policy tuning or management directives will make security "take care of itself."
- The importance of the security policy to the individual and the organization? What are the enforcement mechanisms? For example, in some organizations a person might be casually reminded about security violations. In other organizations he might be dismissed immediately for the same violation.

1.2 Who Needs Security?

"Security" often invokes images of industrial espionage and "hackers." This is a very small element of security. The major reasons for computer security include:

1. Errors. A keying error may delete the wrong file, for example. Secured files and a good backup procedure (which is also an element of security) will reduce these problems.
2. Disgruntled employees (or ex-employees). These may be after confidential information or may wish to sabotage the system.
3. Curious employees. For example, everyone would like to read (and perhaps change) payroll and personnel files.
4. Challenged employees. Breaking a security system is a diverting exercise for some people. No harm is intended. However, if confidential information is to remain confidential, it must be guarded.

There may be legal considerations involved with system security. If *due professional care* is not exercised, an organization might be liable for losses caused by poor computer security. The AIX security features and practices discussed here, when properly used, represent *generally accepted security practices* suitable for a basic commercial UNIX installation.

Several countries have laws about "databases" and security. Your system, as a whole, might be considered a database. These laws may cover anything that might be considered ledger data, information that might influence stock prices, personnel information, and so forth.

Security must be reasonably consistent across an organization. There is no point in securing a file on one system when a copy of the file can be openly read on another system. Again, security begins with top management and the policy (goals) should be understood throughout the organization.

---

[2] Corrupted data is usually much more damaging than missing data. The corrupted data might be used for long periods before errors are detected.

1.3 How Much Security?

Too much security may be as bad as too little security. Too much security, if poorly implemented and administered, may not be effective and can make a system very difficult to use. Increasing system security requires increasing administrative time and effort. Computer security levels defined by the U.S. Department of Defense have become a point of reference for purchasing systems. These levels, in increasing order, are D, C1, C2, B1, B2, B3, and A. Level ″D″ has no certified security functions. The ″C″ levels have discretionary security controls. That is, the user decides which resources to protect and controls (to some extent) how the protection is applied. The "B" levels have mandatory security controls (along with other additional functions). The controls are automatically applied by the system.[3] The ″A″ level is very unusual and only a few examples exist. (These levels are in the context of isolated systems and generally ignore networking issues.)

Installing a ″B″ level system does not automatically answer security concerns. Considerably more planning and administrative time may be needed to administer a ″B″ system than a lower-level system. A system certified for "B" level probably will not run at this level when using only the default security options. Continuing administrative effort is needed to establish and maintain the "B" level functions. **Do not purchase or install more security than you plan to administer.** Administration is not free, and is not automatic.

1.4 System Categories

This document considers two categories of AIX systems: *workstations* (sometimes called *clients*) and *servers*. These terms, workstation, client, and server, can be misleading and may not accurately describe their systems. However, the terms have become very common and will be used here.

A *workstation* (as the term is commonly used) is conceptually a single-user system. The prototype is a desktop machine with a large graphics display. It might also be a desk-side system, and might sometimes have multiple users connected through **telnet** or **ftp**. (X terminals are not considered workstations by these definitions.) A workstation might be owned and used by one person, or might be used (at different times) by a group of people. In general, all the direct (not through TCP/IP) users of a workstation know the *root* password, and all can be considered to be system administrators.

A *server* is a multiuser system, with a defined system administrator. The general users of the system do not know the *root* password. Users may be connected by direct or dial-in ASCII terminals, by telnet or X server connections, or by various other client-server links.

These categories have somewhat different security requirements. We will try to address both categories throughout this document. (The *root password* used in these definitions will be discussed, in detail, later in the document.)

[3] A more complete description of these levels is given in Appendix A, "DoD Classes" on page 105.

1.5 Common Security Exposures

This document discusses many security elements of AIX. However there are three particular elements that are more important than all the others together. These are poor passwords, "set userid" programs, and poor directory permissions.

The problem of poor passwords is not unique to AIX. Every computer user is confident that he can select good passwords. He is usually wrong. Most attacks on UNIX systems are by password guessing. These frequently succeed because it takes only one userid with a poor password to provide a path for gaining unauthorized access to a system. Password quality is discussed in Chapter 3, "User Accounts" on page 13.

Once logged into a system, an intruder[4] often uses "set userid" (usually written as *suid*) programs to obtain wider access to the system. This problem is discussed in 4.2.2, "Permission Bits (Advanced)" on page 49. (By itself, the *suid* function is not an exposure; it is a necessary part of UNIX. It is misuse of *suid* that creates problems.) AIX does not support *suid* for shell scripts. This is a major change from many UNIX systems and is a definite security enhancement. However, it may surprise and cause some disagreement from knowledgeable users.

File security (including files that are executable programs) is generally controlled by *permission bits*, although other controls are available. In fact, two quite different uses of permission bits are required to secure a file. The file, itself, has permission bits. Each level of the directory chain leading to the file also has permission bits. Users who are careful with permission bits for their files, are often careless (or unaware) that the directory permission bits are also important for file security. This exposure is discussed in 4.2.2, "Permission Bits (Advanced)" on page 49.

These three problems (poor passwords, *suid* programs, and directory permissions) account for many common UNIX security problems. Everyone has read about exotic and sophisticated attacks on systems. These exist but they are rare and require unusual skills. This document does not attempt to address the more exotic security problems. It concentrates on common and routine security elements for "normal" AIX systems in "normal" commercial environments.

1.6 Physical Security

Physical security has several aspects, including:

- Access to the processor. Some systems are small enough to be carried away by one person. Some processors have special switches that can be used to bypass normal security.
- Cable security. LANs, in particular, are very vulnerable to unauthorized monitoring.
- Access to privileged terminals such as ″operator consoles″.

[4] Remember, an "intruder" may be a regular employee, with allowed access to the system. When the employee attempts to bypass security, he becomes an intruder.

- Access to diskette or tape drives. These might be used to copy large amounts of data for analysis elsewhere. Also, it may be possible to "boot" an alternate operating system that ignores normal system security controls.

Problems of physical security have been discussed in many documents and these discussions will not be repeated here. However, one element of RISC System/6000 physical security should be understood. This is the keyswitch present on every system. The *normal* position of the keyswitch prevents the system from booting from CD-ROM, diskette, or tape (when the system is setup "normally"). The *secure* position prevents booting from any device and disables the reset pushbutton.[5] This is an important consideration. The maintenance operating system that is used to install the main AIX system can be booted from tape or CD-ROM. This small maintenance system has only the *root* user (and no password is required). All the files on the main disks can be mounted and accessed by the maintenance operating system.

The keyswitch (in the *normal* or *secure* position) prevents booting from CD-ROM or tape. An attended system, such as a workstation, would leave the switch in the *normal* position. An unattended server would normally have the switch in the *secure* position, unless the system is in a secure area. If the *secure* position is used, the key should be readily available to the persons who normally boot the system. The *service* position is used to boot from tape or diskette; this is be necessary when installing a new operating system and when running certain diagnostic programs. In the *service* position, anyone with access to the system can boot a maintenance system that bypasses all security and can access any file on the disks.

	KEYSWITCH POSITION		
Function	Normal	Service	Secure
reset button	E	E	D
boot harddisk	E	D	D
boot tape/diskette	D	E	D

(Enabled or Disabled)

We strongly recommend that the keyswitch be set to normal or secure, and the key removed from a RISC System/6000. The key must be available when it is needed for software or hardware maintenance.

This policy will not prevent someone from opening the processor box and bypassing the keyswitch. In some system models the switch contacts are not readily accessible, and several contacts must be manipulated in a way that is not immediately obvious. In other models, bypassing the switch is fairly easy. In most models the keyswitch, in "normal" position, locks the covers and an intruder may need to damage the system somewhat to obtain access to the switch contacts.

Theft is a problem with personal computers, and the small RISC System/6000s are about the same size as many personal computers. There is no particular

[5] IBM documentation tends to use the terms boot and IPL interchangeably. IPL means Initial Program Load and is normal mainframe terminology for booting an operating system. Also, login and logon, and logout and logoff are used interchangeably.

solution for this problem. Although they may be unattractive, some of the PC-oriented bolt/chain/cable products[6] might be considered in some situations.

Physical access to terminals is not a particular problem if elementary rules about passwords are observed and if users log out before leaving their terminals. **Installation management must have and enforce a policy about leaving logged-in terminals unattended.**

Normal RISC System/6000 installations do not have any privileged terminals, such as operator consoles or maintenance consoles. These functions can be performed (if needed) from any terminal, and depend on the authority of the user (via the userid and password) and not on the particular terminal.[7]

1.6.1 ″Power On″ Hours

Should the RISC System/6000 be powered-up continuously or should it be powered-off every evening? Physical security is only one consideration for this question. AIX normally schedules several tasks for early morning (when there should be very few users). These tasks process daily accounting and remove unnecessary files from the disks. If the system is shut down every evening, these tasks must be started manually or rescheduled in some way.

Unless there is a good reason otherwise, we recommend leaving servers running continuously. Some installations reboot the system once every week or so.[8] There is no need to leave *display terminals* powered on when not in use.

We have no strong recommendations for workstations. Economics and “green” environmental policies may encourage users to turn off their systems at the end of the day. If this is done, the cleanup jobs automatically run in the early morning hours should be changed to run during the day.

1.7 System Administrator

This document frequently refers to the system administrator. For simplicity, we will define a system administrator as a person who knows the *root* password.[9] A server should have a designated system administrator. A workstation (even if shared among several users) probably will not have a formal system administrator, but it will have a lead user who performs the same functions. (Some of these tasks can be done by any member of the *system* group, without knowing the *root* password.)

The administrator normally:

- Installs AIX, AIX updates, and other program products
- Installs new hardware devices
- Adds and deletes users

[6] Examples are listed in almost any personal computer accessories catalog and in the trade press.

[7] The user can configure AIX to restrict certain users to certain terminals. For example, the *root* user might be restricted to a particular console. Clever use of other configuration files could extend selected privileges to certain terminals, but this is not the default or normal situation.

[8] UNIX systems sometimes accumulate dead processes that are not properly flushed from the system. Also, a reboot will remove any virtual storage fragmentation.

[9] This is a very general statement, of course, and ignores administrators within groups.

- Takes backups
- Maintains TCP/IP and other communication services controls
- Monitors and maintains disk space
- Generally monitors the well-being of the system

Generally accepted security practices require that the system administrator devotes some time to security checks. The time required is not necessarily long, but it should be properly planned and consistently used. Recommendations are found throughout this document.

1.8 Computer Security Audits and Reviews

You may be asked to perform or participate in a security *audit* or *review*. This will usually involve demonstrating that reasonable procedures are in place to manage security functions. Specific checks are discussed in the last chapter of this document.

The number of computer security audits performed yearly is growing very rapidly. These are often done in conjunction with a corporate financial audit. Security audits started with large mainframe systems, but now often include smaller systems and networks. The auditors may be “internal” (that is, employees of the organization) or “external” (from an outside organization that specializes in auditing).

A computer security audit does not normally involve detailed program analysis,[10] although a major technical review may involve some program analysis. The basic goal is to verify that “generally accepted security practices” are in effect and are intelligently used. The audit normally checks for major security exposures rather than subtle and exotic holes in the security controls of a system.

The initial requirement for security audits arose as a part of the general financial audit procedure. In the last few years there have also been direct requests from executive management in large organizations. There are several reasons behind executive requests for computer security audits:

- Systems are no longer concentrated in the “IS shop,” behind locked doors. Responsibility for systems has spread throughout the organization, and no one is responsible for overall security.
- Various lawsuits have made executive and director liability an important consideration.
- System administrators are sometimes too close to their own systems and do not see security exposures that may be obvious to an external reviewer.
- Computer security sometimes requires a specialized point of view and specialized skills. These may not be readily available from internal resources.
- System managers (and internal ″reviewers″ or ″auditors″) are influenced by organizational politics, history, and culture. External reviewers should have less bias in their reports.

[10] It may sometimes involve spot checking a few programs or a detailed analysis of a somewhat randomly chosen program.

Security monitoring can be considered a subset of an audit or review. It tends to be an automated process, using various program tools provided by the system vendor or a third party. The intention is to maintain baseline security controls. With some planning, security monitoring can be done remotely.

Chapter 2. AIX Security Structure

UNIX systems, including AIX, do not have a central security manager.[11] The security mechanisms are integral to the AIX operating system. As distributed, AIX is designed to meet the Department of Defense (DoD) 5200.20-STD requirement for C2 level security, including authentication, access control, audit, and object reuse control.

Key elements of AIX security include:

1. Physical access to the system.

   Physical access to a processor (including access to a key, if used by the hardware) can permit someone to start his own operating system. This system might bypass any software security controls and access any files found in the system. Aspects of this are discussed in 1.6, "Physical Security" on page 4 and 3.9.1, "Repairing the root Userid" on page 36.

2. *System state* and *problem state* operation.

   All modern processors have at least two hardware-enforced states of operation. A program operating in *system state* can perform any CPU instruction, alter any register, and so forth. A program operating in *problem state* can use a limited set of instructions and cannot access some of the control registers of the processor. Programs operating in system state can bypass any security controls, while programs in problem state are bounded by restrictions and the environment set by the operating system.

   System state programs establish address spaces of virtual memory and prevent different processes from accessing each other's memory. The same mechanisms prevent programs from changing control bits in kernel structures. For example, an application program cannot manipulate bits in the kernel to permit itself to operate in system state, or to change from a normal user program to a *root* program.

3. *Kernel mode* and *user mode*. The sense of *system* and *problem* states is not extended to the UNIX programming interface. UNIX does provide the concept of *kernel mode* and *user mode* levels, and these very roughly correspond to system and problem states. You can add additional programs to your UNIX kernel. AIX makes this easy by using dynamic linking to load kernel modules. Older UNIX technology required rebuilding the kernel in order to add new modules.

   Kernel extensions, in general, are not subject to normal UNIX security.[12] If you add kernel extensions, you are exposed to whatever these extensions do. Obviously, you should be careful about what is added to your kernel. (The IBM program products that add kernel extensions are "security conscious").

4. Programs operating with *root* authority.

---

[11] RACF, for example, is a central security manager for MVS systems. MVS system functions call RACF at appropriate times to determine if a certain user is permitted to perform a certain function. Application programs can also call RACF to determine if access to a specific resource is permitted for the current user.

[12] This can become a very complex area. Kernel extensions can be written that comply with all security controls. Kernel extensions can be written that ignore all security, operate in system state, and change whatever they wish. In order to install a kernel extension, *root* authority is needed.

© Copyright IBM Corp. 1994

Under UNIX (including AIX), programs operating with *root* authority automatically skip many security checks. Ignoring file permission bits is the most apparent effect. This operation of *root* is part of the defined UNIX interface. Programs operating as *root* are not the same as kernel-mode programs. Except for special cases in the kernel, *root* programs are in user mode (which is in CPU problem state). Later sections of this document discuss *root* authority, beginning with 3.1.1, "The root User" on page 14.

Programs operate with *root* authority because they are called by another process that is operating with *root* authority, or because they *suid* to *root*. This is described in 4.2.2, "Permission Bits (Advanced)" on page 49.

5. File and directory authorization.

   All UNIX systems have *permission bits* to control access to files and directories. AIX (and some other UNIX systems) extend this control with Access Control Lists (ACLs) to provide finer-grain control. File and directory modules (open, close, read, write, and so forth) contain code to check the permission bits and ACLs, as appropriate for the function being performed. These modules skip the checks for *root* users.

   UNIX extends the notion of files to practically everything in the system. Devices are files; system memory itself can be regarded as a file. Permission bits can protect all these resources.

6. Login authentication.

   The login process establishes the user's identity. Several programs perform this function, such as **login, telnet, ftp, rlogin,** and so forth. These programs use *root* authority to establish the new user's identity. The login function is not always used. Any *root* program can assume the identity of any defined user, effectively skipping the login process for that user.[13]

   AIX 4.1 provides excellent login controls. These are discussed in 3.3, "Users" on page 16. You (the administrator) should ensure these controls are used. The login controls are effective only if they cannot be bypassed. This is the reason you must exercise considerable care in permitting *suid root* to be added to your system.

7. Security checks by subsystems.

   These are simply cases of *root* programs being used for a specific purpose. For example, the **telnet** function consists of multiple elements (TCP/IP, sockets, **inetd**, **telnetd**, and so forth) all operating with *root* authority. One of the *root* programs authenticates the user (with **login**, or with ***.netrc*** files) and finally creates a new process running under the end-user's identity.

8. Security aids.

   AIX provides additional security aids. (All UNIX systems provide additional security aids. Except for accounting (a standardized UNIX facility), different UNIX implementations have different sets of security aids.) Some of these are *kernel mode* functions (such as some auditing functions) and some are normal programs running with *root* authority.

Programs that form the kernel and programs that operate as *root* must be trusted. AIX identifies a *Trusted Computing Base (TCB)* subset of the system. Extra utilities are provided to inspect the files that form the TCB and detect

[13] This may happen when .rhost files are used, for example.

changes. Special functions may be provided to ensure that a user is working with a TCB program and not a counterfeit. These elements are discussed in Chapter 7, “Trusted Computing Base” on page 77

The remainder of this document discusses specific security features of AIX 4.1, and how they are used.

2.1 smit

AIX includes **smit**, a system management interface tool. You will use **smit** for many purposes -- both for administration and routine user functions. Any user may use **smit**, although most functions will succeed only if the current user has *root* authority. There is no requirement to use **smit**; any function that can be done with **smit** can also be done with “lower level” commands. **smit** generates these “lower level” commands and depends on them to perform the services requested by the user.

smit provides an architected interface that can be used by many AIX components. For example, DCE and CICS administration is performed through **smit**.[14] One subsection of **smit** deals with systems management, and a subsection of systems management deals with security and users. Using an indented notation to indicate sublevels within **smit**, this can be written:

```
smit
  System management
    Security and Users
      (various lower level menus)
```

The *Security and Users* menus of **smit** cover all normal administrative needs for basic user administration. The next chapter discusses this in detail. Note that **smit** does not cover all areas of security. It concentrates on the direct aspects of user (and group) management. The following diagram of **smit** menus for Security and Users provides a good overview of what user administrative functions can be done through **smit**:

```
smit
  System Management
    Security and Users
      Users
        Add a User
        Change User's Password
        Change/Show Characteristics of a User
        Lock/Unlock a User's Account
        Remove a User
        List all users
      Groups
        List all Groups
        Add a Group
        Change/Show Characteristics of a Group
        Remove a Group
      Passwords
         Change a User's Password
         Change/Show Password Attributes for a User
      Login Controls
```

[14] This means that menus and panels have been added to **smit** to generate various DCE and CICS administrative commands.

```
Change/Show Login Attributes for a User
Change/Show Login Attributes for a Port
```

smit runs in character mode (started from the command line) or in graphics mode under X Windows. The X Windows mode does not use icons, and is quite similar to the basic character mode. We suggest you use both modes. If you are already in X Windows at the time you need **smit**, start it in a window. If you are in line mode (in a shell), we suggest you start **smit** in line mode; there is no need to start X Windows just to run **smit**. The two forms are sufficiently similar that using both does not create any confusion.

2.2 Visual System Manager

AIX provides an alternative to **smit** named VSM, for Visual System Manager. Elements of VSM are started with commands such as **xuserm, xlvm, xmaintm, xinstallm**, and **xdevicem**. VSM, which runs only under X Windows, has an attractive, icon-oriented graphics user interface. VSM provides an excellent demonstration of GUI functions, and may be useful for new administrators (or nonadministrators who must manage a system somehow).

We expect that experienced administrators will find **smit** somewhat faster to work with, and all the descriptions in this document assume the use of **smit**. Many actions described for **smit** can be performed through VSM, but the interface is quite different.

Chapter 3. User Accounts

Managing user accounts is the most routine function of a system administrator. “Traditional” UNIX systems required the administrator to edit several files (such as ***/etc/passwd*** and ***/etc/group***) to add or delete a user. Later generations of UNIX provided commands (or shell scripts), such as **mkuser**, for common user administration actions. AIX, and some other modern versions of UNIX, provide higher-level tools for user administration. The AIX tool is **smit**.

It is possible to perform AIX user administration by editing various files, or by using **mkuser** and similar commands.[15] However, we strongly recommend that you use **smit** unless there is a specific problem that **smit** does not cover well.

User management includes the use of shadow files (for better security), various user quotas (for better resource management), login restrictions (for better security), password criteria (for better passwords), and so forth. There are more administrative files that must be updated, with mutually consistent parameters, for user management than there were under the traditional UNIX systems. Using **smit** is easier and much less error-prone than direct editing of all these files, or using lower-level commands. Use **smit**!!

You will sometimes encounter statements that “real UNIX gurus” always manage systems by editing basic files (such as ***/etc/passwd***). This was a reasonably valid statement some years ago. It is not reasonable today.

Note: This chapter ignores the effect of NIS (previously called “yellow pages”) usage. The following discussion assumes administration is performed directly on the system. NIS is discussed briefly in Chapter 5, “Network Security” on page 61.

3.1 User Identification, UID

A user has two forms of identification in basic UNIX systems.[16] He has a user name, such as *joe*, and a user identification number, or UID. For example, *joe* may have UID 201. All internal system identification uses the UID. The *only* place where the user name (or “login name,” or “login userid”) is used is the ***/etc/passwd*** file and its closely associated administration files such as ***/etc/group*** and (in AIX) the ***/etc/security/...*** shadow and configuration files.

Every user should have a unique UID. If **smit** is used to create new users, it will automatically give every user a unique UID. Although you can override this by forcing a UID number in **smit** or by editing the ***/etc/passwd*** file, such actions should be strongly discouraged. If two users have the same UID, then they are the same user for all practical purposes. A reasonable security policy should prohibit assigning the same UID to multiple individuals. If you want C2-level security, *you must not permit shared UIDs*, because this defeats accountability and auditability controls.

[15] AIX has a large number of **mk...**, **ch...**, and **rm...** commands. Examples are **mkuser**, **chuser**, and **rmuser**. These commands are intended for use by **smit**, although you can use them directly if you wish.

[16] He may have more than two if additions such as DCE, CICS, RDBMS, and other subsystems are considered.

© Copyright IBM Corp. 1994

You will encounter some arguments for sharing UIDs. This usually occurs in the context of a particular application. The argument goes like this:

1. Each user (sharing a common UID) must still login under his own name and must enter his unique password. This removes any need to share passwords.
2. The application and its data files can be made more secure by limiting access to only the owner, and the "owner" is the shared UID. This allows only users with the shared UID to access the files.
3. The application program, by internal design and processing, provides all necessary security and can differentiate between the various users sharing the common UID.

The problem with this logic is that other applications do not differentiate between all the users with the common UID. Accountability is lost. The desired level of sharing can usually be accomplished with unique UIDs and appropriate *group* definitions and usage. Nevertheless, there may be existing applications that require shared UIDs.[17]

You can check for shared UIDs (whether root's or normal users) by displaying the ***/etc/passwd*** file. The third operand of each line is the UID number. If two users (two lines) contain the same number, then they are sharing a UID. (The fourth operand is a group ID, and it is normal for multiple users to share group IDs.) If a considerable number of users exist, checking UIDs manually could be tedious. The **usrck** command is available to do these checks and is described in "The usrck Command" on page 34.

In this document (and in most UNIX documentation) the term "userid" means an account name (such as *joe*), and "UID" means an internal user number, such as 201.

3.1.1 The root User

All UNIX systems have a user named *root*.[18] The *root* user will be discussed throughout this document. Most security controls do not apply to *root*. He can read and write in any file, for example. (The *root* user is often called the *superuser*, and the **su** command (which is normally considered to mean Switch User) is sometimes referred to as the SuperUser command.)

Many user and system administration functions require you to be *root*. This is because:

1. *root* owns the files being updated, or
2. *root* can write in the files regardless of who owns them.

For practical purposes, a system administrator must be able to run as *root*; the primary system administrator of AIX is the *root* user. For personnel backup, maintaining multiple shift operations, and so forth on larger systems, there will probably be several people who know *root*'s password. This disclosure of a

---

[17] Another use of shared accounts is for training. In this case, many users may know the passwords of a few training accounts. Obviously, these accounts should be restricted to specialized groups, have very restrictive quotas, and so forth. The best suggestion is to limit training accounts to specific systems. Without using special shells, it is difficult to rigorously limit a UNIX account, and you may not want user training to take place on your production systems.

[18] This statement is not quite true. All UNIX systems have a user who is assigned user number zero (UID = 0). By convention, this user is named *root*, although this is not an absolute requirement. It is the UID=0 that provides this user with his special powers.

password is probably necessary (for large systems), but must be managed with considerable care.

It is possible to share UID=0; that is, have multiple login userids which equate to UID 0. This permits several administrators, each with his own password, to have UID 0 (or "root") authority. This situation is not uncommon, but we recommend against it. Among other problems, it encourages routine operation as *root*.

Where multiple administrators are required (each of whom needs to have *root* authority for certain functions), we strongly recommend that each administrator log in with his own userid (which has its own unique UID) and then **su** to *root* when required.[19]

We have two strong recommendations for *root*:

1. Protect the password! Your system has no protection against anyone with *root*'s password.
2. Do not routinely operate as *root*. Switch to it only when needed for an administrative action. This prevents "accidents," which are the source of many security failures in UNIX.

AIX keeps a log of all uses of the **su** command. This log is in ***/var/adm/sulog***. This log can be useful for an overview of how **su** is being used, or for reconstructing a series of events. It logs all use of **su**, including switches to *root*. See Chapter 6, "Logs and Accounting" on page 75 for a discussion of AIX logs.

3.2 Single-user Workstations

"No one else uses my workstation. Why hassle with security?" "It's my system. I login as *root* and never have any security problems." "No one should tell me how to manage *my* system. It's my problem and I'll make my own choices."

You may have different thoughts about these statements depending on whether you (the reader) are the owner of a single workstation or whether you are attempting to administer many workstations throughout a department or enterprise.

Some degree of security is important even for a single-user workstation. As a minimum, you probably want to prevent the wrong people from using the system. Default userids (such as *root*), without passwords, are part of the basic AIX system shipped by IBM. Anyone with physical or network access to the system can use it until you change these defaults.

Most "single-user" workstations are networked, and no system on a network is really single-user. Remember that being attached to a network is a two-way relationship. See Chapter 5, "Network Security" on page 61 for more information. It is foolish to ignore security in any workstation connected to a network.

[19] **su** is a standard command allowing a user to temporarily switch to another user's identity (that is, use another user's UID). If **su** is used with no operand, it attempts to switch to the *root* identity. The **su** command requires the user to enter the password of the target user. Thus anyone obtaining *root* authority through **su** must know root's password.

Even if your system never will be used by anyone other than yourself, never use *root* as your routine userid. Many "security" checks are also "accident prevention" checks. Operating as *root* often overrides the accident-prevention aspects of security. As an extreme example, perhaps you want to display ***/etc/passwd***, but you absent-mindedly enter **rm /etc/passwd** instead of **pg /etc/passwd**. If you are not *root*, you will receive an error message. If you are *root*, you have created a substantial problem for yourself.

We recommend administering workstations as if they were small multiuser systems. That is, define appropriate userids and observe reasonable directory and file security. Significant group definitions might be omitted for workstations (assuming they are not needed for NIS or NFS usage).

3.3 Users

Use **smit** to add, change, and delete users. Do not perform routine user administration by directly editing the various files involved. Do not directly use **mkuser** and related commands for routine user administration.[20]

If you want the default group for new users to be something other than *staff*, you should edit the ***/etc/security/mkuser.default*** file. This file contains the default attributes for users created by the **mkuser** command. **Smit** uses **mkuser** to add users. The file looks like this:

```
user:
          pgrp = staff
          home = /u/$USER
          shell  = /bin/ksh
          auth1 = SYSTEM;$USER
```

Change this so that an appropriate default group is defined by *pgrp*. (A discussion of group definitions begins on 3.5, "Groups" on page 27.) The new group must be defined before you create new users assigned to the group.

You also may want to edit ***/etc/security/.profile*** before adding users. This is the prototype for the user's own profile; this file will be copied to ***$HOME/.profile*** as part of the **smit** process of creating a new user. A user can change his own ***$HOME/.profile*** file, but most users do not. The prototype is used only when a new user is created.

Please give some thought to userids, since they are used by humans. Employee numbers, for example, make poor userids because they are not meaningful to other people. Electronic mail is important on most systems and meaningful userids make handling mail much easier. Users are much happier if their userid is the same on all systems they use. This is easy if the new user already has a established userid in the organization --- just ask him for his userid. If the user is new to the organization, some coordination effort may be required to assign him a userid. (An organization's security standards should contain guidelines for creating userids.)

[20] The system will not prevent you from doing user administration by editing files or using mid-level commands. You may be required to do this in unusual situations. Modern UNIX systems, such as AIX, have more files involved in user administration than earlier UNIX systems, and using **smit** helps ensure consistent updates to the files.

3.3.1 User Parameters in Smit

The following menu is produced by **smit** for adding or changing a user's definitions. Subsets of these same menu items are produced by other **smit** menus. **Read 3.3.2, "System Defaults" on page 20 before deciding how to use elements in the following menu.**

```
smit
 Security and Users
  Users
   ADD a User

    1  * User NAME                        [joe]
    2    User ID                          [   ]
    3    ADMINISTRATIVE User?             false
    4    Primary GROUP                    [staff]
    5    Group SET                        [staff]
    6    ADMINISTRATIVE GROUPS            []
    7    Another user can SU TO USER      true
    8    SU GROUPS                        [ALL]
    9    HOME Directory                   [/usr/guest]
    10   Initial PROGRAM                  []
    11   User INFORMATION                 []
    12   EXPIRATION date (MMDDhhmmyy)     0
    13   Is this user ACCOUNT LOCKED?     false
    14   User can LOGIN?                  true
    15   User can LOGIN REMOTELY?         true
    16   Allowed LOGIN TIMES
    17   Number of FAILED LOGINS before   [0]
             user account is locked
    18   Login AUTHENTICATION GRAMMAR     [compat]
    19   Valid TTYs                       [ALL]
    20   Days WARN USER before pw expires [0]
    21   Password CHECK METHODS           []
    22   Password DICTIONARY FILES        []
    23   Number of PASSWORDS before reuse [0]
    24   WEEKS before password reuse      [0]
    25   Weeks between pw expire & lockout[-1]
    26   Password MAX. AGE                [0]
    27   Password MIN. AGE                [0]
    28   Password MIN. ALPHA characters   [0]
    29   Password MIN. OTHER characters   [0]
    30   Password MAX. REPEATED chars     [0]
    31   Password MIN. DIFFERENT chars    [0]
    32   Password REGISTRY                []
    33   MAX FILE Size                    [2097151]
    34   MAX CPU Time                     [-1]
    35   MAX DATA Segment                 [262144]
    36   MAX STACK Size                   [65536]
    37   MAX CORE File Size               [2048]
    38   File creation UMASK              [22]
    39   AUDIT classes                    []
    40   Trusted path?                    nosak
    41   PRIMARY Authentication Method    [SYSTEM]
    42   SECONDARY Authentication         [NONE]
```

The line numbers (1 through 42) are not shown in the smit display, but are shown here to assist the following discussion. Some of these options are important for security and must be understood.

Enter the userid (line 1, NAME) for the new user. The userid must be unique on a given system. The User ID (line 2) is the UID. The smit process will automatically assign the next UID; you should not override this field without a very good reason. Leave it blank.

The administrative fields (lines 3 and 6) are described later. Leave these fields unchanged (that is, "false" and blank) for normal users.

The User can LOGIN (line 14) field determines whether this user can login directly. (Indirect login is partly managed by lines 7, 8, and 15.) A normal user should be allowed to log into the system. Standard system accounts, such as *bin*, are special cases in which no login should be permitted. Another special case is *root*, which is discussed later. This parameter sets a flag in the ***/etc/security/user*** stanza relating to this user. It does not set an asterisk in the password field in ***/etc/passwd*** to inhibit login.

The SU Groups (line 8), SU TO USER(line 7), and LOGIN REMOTELY (line 15) controls may be used to restrict access to this account. The normal (and default) values are shown above. These default values permit access to the account in several ways.

The SU TO USER field determines whether any other user can switch to this account by using the **su** command. Not even *root* can **su** to the account if this flag is *false*. If SU TO USER is *true* then the SU GROUPS field provides some control over which other users can **su** to this account. Only users who are members of a group listed in this field are permitted to **su** to this account. This field is not effective against *root*, who can **su** to the account regardless of group restrictions. The default value of *ALL* is a keyword meaning all groups; that is, meaning no limitations based on groups. (An exclamation mark preceding a group name functions as a "not" symbol. Members of the named group are not permitted to **su** to this userid.)

The LOGIN REMOTELY field controls access through the **rlogin** and **telnet** facilities of TCP/IP. If set to *true* (the default), anyone (anywhere on a connected TCP/IP network) can log into this account using **telnet** if he knows the account's password. This parameter does not control access through the **ftp** function of TCP/IP.

The initial PROGRAM (line 10) is the name (with the full path) of the program to be given control when this user logs into the system. It is normally the name of a shell, such as ***/usr/ksh***. If this field is blank (the normal case), the initial program name is taken from ***/etc/security/mkuser.default***.

The EXPIRATION date (line 12) is usually left as 0 (meaning no expiration date). The date format is shown in the menu. A typical entry might be "0330000095" (MMDDhhmmyy). This parameter is useful for temporary accounts, such as for visitors or contractors. The account is disabled after the specified date.

The ACCOUNT LOCKED field (line 13) is normally *false*, and can be used as the single point of control to disable a userid. (It will not force a user off the system if he is currently logged in.)

You can limit a user to logging in between certain hours by using LOGIN TIMES (line 16). There are several formats for this parameter, and they are explained in the comments in ***/etc/security/user***. Once logged into the system, a user will not be forced off if his session extends beyond his valid hours.

Valid TTYs (line 19) defines the terminals this account can use. (It does not control pseduo-terminals, as used by **telnet** and other remote connections.) The full path names of terminals must be given, such as /dev/tty1. An exclamation mark before a terminal name means that terminal may not be used. The ALL keyword means that all terminals may be used. The ability to limit specific users to specific terminals can be a strong security tool, but must be used with care. For example, *root* might be limited to the local console, although this leaves an exposure in the event of a hardware problem.

The WARN USER parameter (line 20) causes a warning message to be sent when a user logs into the system if his password is due to expire within the specified period. We recommend using this parameter if you enforce password changing (line 25). A user needs some time to devise a good password, and this parameter prevents an unexpected (when the current password expires) demand for a new password.

The AUTHENTICATION GRAMMAR (line 18) should be left with the default "compat" value. This will be used with future enhancements for distributed systems.

Various password controls (lines 21 - 31) are discussed in 3.3.4, "Passwords" on page 21. We suggest you leave these lines unchanged.

Do not enter anything in the REGISTRY (line 32). This is for future use with DCE or other remote registry functions.

The process limitations (lines 33 through 37) provide some protection against runaway programs. The Max FILE size (line 33) specifies an upper bound for the *ulimit* parameter. The *ulimit* is the maximum size (in units of 512 bytes) of a file written by this user. The user can change the value with the **ulimit** command, but cannot exceed the value set in the **smit** field. It appears the minimum value accepted by **smit** is 8192. Note that this maximum file size is for a single file; it does not limit the total amount of disk space consumed by this user.

The CPU Time parameter (line 34) limits the maximum running time of any single program. The units are in seconds and is the AIX "process" time, which is often more than the pure CPU time. When a process exceeds this time limit, it is interrupted by AIX. The user sees an error message and a new shell prompt.

The UMASK (line 38) is the default *umask* for the user. A user can change his *umask* value, for the duration of a session, with the **umask** command. (The *umask* function is discussed in 4.2.3, "The umask Variable" on page 52.)

Do not change lines 39 - 42 (AUDIT, Trusted path, PRIMARY, SECONDARY) unless you have specific requirements. Each of these topics is discussed later in this document.

The results of defining a new user (using the **smit** panel shown above) are additions to several files:

1. ***/etc/passwd*** will contain a new line defining the user.
2. ***/etc/security/passwd*** will contain a new stanza for the user's encrypted password and a few flags.
3. ***/etc/security/user*** will contain a new stanza containing some of the user's restrictions.

4. ***/etc/group*** will be altered to add the new user to one or more groups.
5. ***/etc/security/limits*** will contain a new stanza containing some of the user's environmental limits.
6. ***/etc/security/.ids*** will be updated to contain the next available UID.
7. ***/home*** will contain a new directory, which is the home directory for the new user. (This assumes "normal" definitions in the smit panel.)

3.3.2 System Defaults

Many of the menu items listed above might be best set as *default* values, instead of individual user values. For example, you might want to establish a maximum password age (line 26 above) for all users rather than setting it for each user.

Most of the user parameters are stored in ***/etc/security/user***. You should display this file (with an editor or the **pg** command) and study its format. There is a stanza for every user defined in the system and a "default" stanza. As *root*, you can edit the default stanza (using **vi** or your favorite editor) and change the default values.[21] Changes are effective immediately, the next time a user logs into the system, unless that user has overriding values in his stanza. Also, parameters in the default stanza will appear as default values the next time you add a user with **smit**.

Of the 42 menu items displayed by the smit add/change user menus, 30 are stored in ***/etc/security/user*** and can be controlled by values in the default stanza. The ***/etc/security/limits*** file is organized in the same way, and defaults for the six user limits (max file size and so forth) can be set here.

The advantages of setting default values instead of many individual user values are obvious. The default values can be changed easily. Individual user values should be specified only they need to be different than the default values. The ***/etc/security/user*** and ***/etc/security/limits*** files are used whenever a user logs into the system; changes will not affect a user who is currently logged into the system.

3.3.3 Shadow Files

Traditional UNIX used the ***/etc/passwd*** file in a variety of ways. A user was created by adding a line to this file. (This is still true in modern UNIX systems.) The user's password, in an encrypted form, was stored in the line. Other key parameters, such as the user's UID, default group, his home directory, and his initial program (normally a shell) are specified in this line.

The lines in ***/etc/passwd*** are used by many programs to translate between a UID (the internal numeric identification of a user) and a userid (the external identification of the user). For this reason, ***/etc/passwd*** must be readable by any program and any user. That is, it must be "world readable." This means that all user passwords, in their encrypted forms, are readable by anyone. The encrypted passwords are exposed for any user to examine.

[21] There is no **smit** menu to alter the "default" stanza. All the other (user) stanzas can be changed by using **smit**. If you use VSM (the Visual System Manager), a default user "template" can be established, but the default values are not stored in ***/etc/security/user***. A template value only affects users added after the template value changes.

The standard UNIX password encryption scheme (used by practically all UNIX systems, including AIX) was considered very good when originally designed. Increased processor speeds have helped create exposures and future processor speeds will make the situation worse. The current exposure is based on guessing passwords. By hand, this is a slow process. When automated, based on large dictionaries of likely passwords (and their permutations), a password guessing program will often succeed in finding passwords for several users in any larger UNIX system. These programs are discussed more in 3.9.2, “Password Cracker Programs” on page 37.

Password cracking programs require access to the encrypted passwords. If the encrypted passwords are in the ***/etc/passwd*** file, they are readily available to anyone. The solution has been to (optionally) move the encrypted passwords to another file. The “other” file is generally called a *shadow* file, and (for AIX) is ***/etc/security/passwd***.

A line in ***/etc/passwd*** (which is readable by anyone) can have four types of entries in the password field (the second field in a line):

1. A null entry. (Fields are separated with colons. A line beginning *guest::201:: ..* is for userid *guest*, has a null password field, has UID 201, and so forth.) A null password field means no password is required for this userid.
2. An asterisk. This is one way to disable a userid. (AIX normally uses another field in ***/etc/security/user*** to disable a user, but uses the asterisk method when a user is first defined.)
3. An exclamation mark. This means that the encrypted password is in the shadow file ***/etc/security/passwd***. This is the normal case for AIX.
4. An encrypted password. An encrypted password is always 13 characters long.

Some accounts can have encrypted passwords in ***/etc/passwd*** and other accounts can have their encrypted passwords in ***/etc/security/passwd***. AIX will work properly with both cases, but we strongly recommend against placing encrypted passwords in ***/etc/passwd***. **smit** and the **passwd** command will automatically place an exclamation mark in ***/etc/passwd*** and place the encrypted password in ***/etc/security/passwd***. There are several other files in the ***/etc/security*** directory. These are all related to security controls and are sometimes called the “security shadow files.”

3.3.4 Passwords

When a new user is added with **smit**, the account is automatically disabled by placing an asterisk in the second field (the password field) of the ***/etc/passwd*** line for the new user. The administrator (working as *root*) must use **smit** or the **passwd** command to set an initial password for the user. Because the password was set by an administrative user (*root*), the new user will be asked to change it the first time he logs into the system. (The ADMCHG flag in the user′s entry in ***/etc/security/passwd*** indicates a password change is required.) Setting the initial password enables the new account, permitting the new user to log into the system.

The **passwd** command is the “normal” UNIX command for changing passwords. The command can be used by any user to change his own password, or by *root* to change any user′s password. There is no particular advantage to using **smit** to change a password; the **passwd** command does the same thing.

The **smit** function for changing a password is in the same menu as the function for adding a new user. It is usually convenient for the administrator to set an initial password for a new user immediately after he creates the new user, and the **smit** function is convenient then. In some cases, it may be appropriate to create a number of new users but not enable them (that is, not assign initial passwords). For example, a new group of student userids might be created at convenient times but not enabled until their class begins. In this case, using the **passwd** command may be more convenient than setting the passwords through **smit**.

Remember that an administrator (running as *root*) must always assign an initial password in order to activate a new account. A new account cannot be used until this is done.[22] There is a well-known exposure here. What password should the administrator set as the initial password? Administrators tend to set a common password, such as the userid or a department name, for all new users. Knowing this, anyone can "steal" a new account by being the first to log into the account (using the standard initial password). There are two solutions for this problem:

1. The administrator can set an obscure password (different for every new user) and inform the user of the selected password.
2. The administrator can delay setting the initial password until the new user is ready to log in. This means there will be a short period when a standard initial password is exposed.

In either case, the new user is prompted to alter his password as soon as he logs into the system.

Many security failures begin with poor passwords. Password quality has been discussed many times in many places; these discussions will not be repeated here. The following are basic guidelines:[23]

- **Do not** use your userid or any permutation of it.
- **If** you use the same password on more than one system, be extra careful with it. Never use the same *root* password on multiple systems.
- **Do not** use any person′s name.
- **Do not** use words that can be found in the online spelling-check dictionary, especially for a networked or larger multiuser system.
- **Do not** use passwords shorter than five or six characters.
- **Do not** use swear words or obscene words; these are among the first words tried when guessing passwords.
- **Do** use passwords that you can remember. Do not write down your password.
- **Do** consider passwords that consist of letters and numbers.
- **Do** use passwords that you can type quickly.
- Two words, with a number in between, make a good password.
- A word (with at least six characters), with a numeric digit inserted in the word, is an excellent password. (But do not form the digit by changing an

---

[22] This statement assumes the account was created, in the normal way, through **smit**.

[23] The new password quality controls in AIX 4.1 can help enforce these guidelines.

"l" to "1" or an "o" to "0.") A word with an internal digit is a better password than a word with a leading or trailing digit.

- A pronounceable password is easier to remember.
- AIX checks only the first eight characters of the password; however the word can be longer than eight characters.

You can specify password quality and composition rules for each user, or by editing the default stanza in ***/etc/security/user***. Individual controls can be set using the **smit** menu discussed in 3.3.1, "User Parameters in Smit" on page 17. The controls are:

```
                 recommended   default
     minage           0            0      (weeks. Use 0)
     maxage          12            0      (maximum age in weeks)
     maxexpired       4            0      (weeks after expire)
     minalpha         1            0      (alpha characters)
     minother         1            0      (non-alpha characters)
     minlen           6            0      (minimum length)
     mindiff          3            0      (different from last pw)
     maxrepeats       3            8      (repeated characters)
     histexpire      26            0      (prohibit reuse, weeks)
     histsize         8            0      (number of old passwords)
     pwdwarntime     14            0      (warning time, days)
```

The default values provide no password quality controls. This is intentional, as it conforms with the expected characteristic of "standard" UNIX.

maxage/minage defines the maximum/minimum age (in weeks) of a password. The default is 0 in both lines, indicating that the password has no minimum or maximum age. We recommend that you do not use the *minage* parameter. It can create awkward situations and may cause more trouble than it cures. Consider using smaller *maxage* values for privileged users such as *root* and members of the *system* group. The *maxage* of a password limits the time period during which an exposed (or disclosed) password can be used.

There has been some debate whether a rigid password expiration period is a good option. If a user suddenly must select a new password, he may select a trivial or poor password. That is, he may not be prepared to seriously think about a new password while he is trying to log into the system for some other purpose. The *pwdwarntime* parameter (specified in days) causes AIX to warn the user shortly before his password expires. This permits the user to change his password in a timely manner.

The *maxrepeat, mindiff, minlen, minalpha*, and *minother* parameters provide basic quality controls. These control the maximum number of repeated characters, the minimum number of characters that a new password must differ from the previous one, the minimum length, the minimum number of alphabetic characters, and the minimum number of nonalphabetic characters that must appear in a password.

AIX has an option to remember old passwords. (The ***/etc/security/pwdhist.dir*** and ***/etc/security/pwdhist.pag*** files are used.) The *histexpire* parameter specifies the number of weeks that must elapse before a password can be reused. The *histsize* parameter specifies the number of other passwords that must be used before a given password can be used again.

AIX provides two more password quality control functions. A file of invalid passwords can be specified (with the *dictionlist=* parameter) and a list of user programs can be specified (with the *pwdchecks=* parameter) to perform any customized password checking. The file of invalid passwords might be ***/usr/share/dict/words*** (if it is present in your system) or any other list of words. These parameters can be set for individual users through **smit** or set as defaults by editing ***/etc/security/user***.

3.4 Search PATH For User

The PATH is an environmental variable used by the current shell when searching for executable files (commands).[24] When using a normal shell, a user can change his PATH specification at any time. There is no reasonable way to prevent changes. (The restricted shell, discussed in Chapter 7, "Trusted Computing Base" on page 77 does not permit changes to PATH.)

One security goal is to prevent *root* (or any other user, for that matter) from executing a counterfeit program. For example, if ***/tmp*** (an unprotected directory) is the first element in *PATH*, and if someone places a program named **su** in ***/tmp***, then this **su** will be executed instead of the correct system **su** program. The *PATH* exposure is a simple concept, and you (the system administrator) must understand it. You should devote whatever time is necessary to understand it. You cannot hope to maintain a secure system if you do not understand *PATH* handling.

A user′s *PATH* is normally set (using the system profile and the user′s profile (if it exists)) when he logs into the system. Both ***/etc/profile*** and ***$HOME/.profile*** are executed automatically when a user logs into the system.[25] The *root* user often has the root directory as his home directory, and ***/.profile*** (if it exists) will be executed when *root* logs into the system.[26]

When switching user identities with the **su** command, the target user′s profile is not automatically executed. (Using a "-" flag with the **su** command will cause the target user′s profile to be executed, but this may have after-effects on the current user after exiting from the target user′s identity. Typically, the "-" flag is not used with **su**.) For example, if you log in as a normal user and then **su** to *root*, you continue to use the profile (and PATH) established by the original user identity. This can be the source of serious exposures, similar to this:

1. A user (wanting to obtain *root*′s password) writes a small C program to counterfeit the initial appearance of the **su** command. That is, it asks for a password.
2. The user compiles and links this program into his home library.
3. The user alters his PATH to search his home directory first, before looking in various system directories.
4. The user asks the administrator for help with a problem that is likely to require *root* access.

[24] Unlike DOS, OS/2, and Windows, UNIX systems do not automatically search the current directory when looking for an executable file.

[25] Note that the profile file name in the user′s home directory begins with a period.

[26] Root′s profile may be marked non-executable. This does not prevent the login process from executing it.

5. The administrator sits at the user′s workstation and uses **su** to switch to *root*. When the administrator enters the **su** command, the system searches the current home directory (as directed by the PATH) and finds the counterfeit **su** program and executes it.
6. The counterfeit program prompts the administrator for the *root* password, stores the password in a hidden file, sends an error message indicating an incorrect password, and erases itself.
7. The administrator thinks he has entered the wrong password and tries again. This time, the correct **su** command obtains control (because the counterfeit program is gone) and the session continues normally.
8. The user later reads the *root* password from the hidden file and is able to login as *root*.

This is the classic Trojan horse attack and it worked because the administrator executed **su** using the wrong PATH. There are two lessons to be taken from this type of attack:

1. An administrator, when executing as *root*, should always enter the full pathname of commands if he is working under another user′s environment. This avoids usage of the existing PATH definition.
2. The PATH for a normal user should search the standard system directories before searching the current directory or specific $HOME directories.

The default path (set by the default user profile) for AIX is:

```
PATH=/usr/bin:/etc:/usr/sbin:/usr/ucb:$HOME/bin:/usr/bin/X11:/sbin:.
```

Subdirectories within ***/usr*** contain most of the AIX commands used by normal users. The ***/etc*** directory contains symbolic links to commands in more remote directories. Notice that the system libraries are searched first. After the system libraries are searched, ***$HOME/bin*** (a conventional location for products installed in the user′s home directory) is searched. The "dot" in the last position of the PATH is significant. This indicates that the current directory should be searched.

Ignoring minor elements (X11 and /sbin), the PATH search order is: system directories, home directory (bin), and current directory. This is a safe search order, although one can argue that the current directory (the "dot") should not be in the PATH at all.

3.4.1 Timeouts

Timeouts are used to automatically log out a terminal that has been inactive too long. The timeout function is provided by AIX shells, not by the basic AIX kernel. By default, there is no timeout period set. The Korn shell uses the TMOUT variable and the Borne shell uses the TIMEOUT variable to provide timeout values. You (the administrator) should set one or both of these variables if you want to automatically log off terminals after an excessive idle period. We strongly recommend using this function because unattended terminals are serious security exposures.

We recommend adding lines similar to:

```
TMOUT=45
TIMEOUT=45
export TMOUT TIMEOUT
```

to ***/etc/profile*** or to ***/etc/security/.profile*** (from where it will be copied to the home profile of new users). The timeout is expressed in minutes. A reasonable timeout value can be the subject of much argument, but will probably be between 20 and 120 minutes.

The timeout period is for the shell. If a user nests several shells (by issuing the **ksh** command repeatedly, for example), the shells will timeout in reverse order with each one taking the full timeout period. The timeout period is a shell or environmental variable and can be changed by each user. You cannot readily enforce a standard value (unless your users are unaware that they can alter the timeout value).

3.4.2 Prompts

You may want to set the shell prompt to show the current directory. For Korn shell users, this is done by adding the following two lines to ***$HOME/.profile***:

```
PS1='$PWD $ '          (use single quotes)
export PS1
```

This change provides a shell prompt with the current path name followed by the traditional "$." Unfortunately, this simple technique provides a misleading prompt if the user executes an **su** to *root*. The "$" will still appear instead of the "#" which is traditional for *root*. This can be avoided by (1) not using this alteration for users who frequently **su** to *root*, or (2) using the command **su -** (with the "-" flag) when changing to *root*.

A prompt displaying the current directory path is helpful for many users, especially if they routinely work with multiple directories. There are no security elements involved (other than the misleading prompt after **su** to *root*), but a more informative prompt may reduce user errors.

There are many more sophisticated methods for obtaining informative prompts. Almost any book discussing UNIX usage or administration will contain suggestions for shell prompts. We recommend the simple version (listed above) only if it fits your needs or if you are not comfortable with (that is, do not understand) more complex prompt functions described elsewhere.

3.4.3 Disabling the root Userid

There is seldom a good reason for logging in as *root*. Most system "accidents" in UNIX are partly caused by routine use of *root* as a working userid. After your system is installed, **you may want to disable the ability to login as** *root*. Authorized users (those who know the password for *root*) could then **su** to *root* after they login under their normal userids.

Disabling *root* is easily done with **smit**:

```
smit
-Security and Users
--Users
---Change / Show Characteristics of a Userid

   * User NAME                         [root]

     ...
     Another user can SU TO USER?      [true]
     ...
     User can LOGIN?                   [false]  <---
```

```
User can LOGIN REMOTELY?        [false]  <---
```

Do not disable *root* by editing ***/etc/passwd*** and changing the password field. This will also prevent you from using *root* through **su** or **telnet**.

3.5 Groups

In larger installations, good system administration usually revolves around *group* definitions. Guidelines for forming groups should be part of any security policy. Defining groups for large systems can be quite complex, and is beyond the scope of this document.[27] Users will often belong to more than one group, but group membership should not be excessive.

If at all possible, group definitions should extend across all system platforms: MVS, UNIX, NetWare, and so forth. That is, a given group name should have the same members, the same security associations, and similar administration on MVS and UNIX and LAN systems. Good group definitions are often related to job functions instead of a strict organizational structures; for example, there may be a group for secretaries, regardless of their department. This is a difficult goal. System administrators will seldom do it voluntarily because it requires endless "coordination" meetings with other system administrators. Nevertheless, it is a good goal because it forms and helps enforce a meaningful security policy for an enterprise.

The standard UNIX file security controls, the *permission bits*, provide very limited granularity. (The AIX *ACL* functions extend this, and are discussed later.) Well-planned use of group definitions substantially extends the usefulness of the permission bits.

However, it must be admitted that most UNIX administrators ignore group definitions, or, at best, define groups for use only within their system. It can be argued that no (or minimal) group definitions are better (more secure, less hassle) than poorly planned group definitions. Poorly planned groups tend to overlap in unexpected (and unsecure) ways, especially if a new group is defined for every new situation.

Our recommendations are:

1. If possible, coordinate group definitions in as large a context as possible. At the enterprise level is best.
2. At whatever level the groups are defined, *think!!!* Consider scenarios. Ask for advice. Once established and in use, *group definitions are exceptionally difficult to change.*

3.5.1 AIX Group Usage and Administration

A user can be a member of multiple groups, and AIX will automatically search all of a user's groups (if necessary) for file access permissions. This is a substantial improvement over some earlier systems. It allows good control with a reasonable number of groups, even for complex organizations.

[27] One goal is to have a moderate and stable number of groups. A symptom of poor security and poor administrative planning is a constantly increasing number of groups.

In some cases, the multiple group search for permissions might lead to unanticipated results. If ACLs are used (see 4.3, "The ACL Commands" on page 53), it is possible that conflicting levels of authority exist for different groups.

The **newgrp** command can be used (by a normal AIX user) to switch his primary group. (He must be defined as a member of a group before he can switch to it, of course.)[28] This may be important when creating files because, by default, the *current* group name is assigned to a newly created file. (An alternate method of determining group ownership of a new file is discussed in 4.2.2, "Permission Bits (Advanced)" on page 49.)

Your most common group name should be made the default group name for new users; as supplied by IBM, the default group name is *staff*. Edit your default group name into stanza *pgrp* in the file ***/usr/lib/security/mkuser.default***. This file provides default values for the **mkuser** command and **smit.**

For example, you might want your default group to be *office*. Edit ***/usr/lib/security/mkuser.default*** and change:

```
user :
          pgrp = staff
```

to:

```
user :
          pgrp = office
```

New users added to the system (using **smit**) will default to group *office*. Of course, you may assign users to specific groups instead of taking the default. (You must create a group before you can assign users to it, of course. Use **smit** to create new groups.)

There are two types of groups in the system: administrative groups and normal groups. An *admin* group is defined in ***/etc/security/group*** by the *admin* stanza. In every group there can be a group-administrator. This is defined in ***/etc/security/group*** by the *adms* stanza.

The administrative parameters are confusing. If the value *admin = true* is in ***/etc/security/group***, then this indicates an administrative group. But *admin=true* in ***/etc/security/user*** means that the user has administrative authority for that specific group which is equal to the *adms* stanza in ***/etc/security/group***. With *admin=true*, the user can administer that group.

The administrative group and authority has very little effect in AIX.[29] **We recommend you ignore the administrative groups and users for smaller systems.** Smaller systems can use *root* to perform user administration, and *root* does not need any other administrative authority in AIX. Large systems (with more than 30 or 40 users) might need group administrators. If your security policy permits it, the easiest way to implement group administrators is to allow them to **su** to *root*. That is, give them the password for *root*. If your security policy does not permit group administrators to know the *root* password, then you might use the *admin* group and attributes.

---

[28] Some traditional UNIX systems permitted a user to switch to a group in which he was not a member, if he supplied the password of the group. AIX does not support this function, and does not support any use of group passwords.

[29] This may change in future releases, so do not abuse the administration definition by making everyone an administrator.

AIX does not implement or use group passwords. It is not possible to log in using a group name.[30]

IBM provides a group named *security*. Any member of this group can read all the user-administration files in the ***/etc/security*** directory, and can execute many of the system administration commands. With little effort, a member of the *security* group can gain *root* authority; therefore only trusted personnel should be in this group.

Group Usage for Workstations

A workstation *may* be a special case for group definitions. A workstation that is used only by a small number of users (or only one user) and that is never the target of **telnet** or **ftp** operations by anyone else is a special case. (It may also be a rare case because most workstations are members of a network and over time, for one reason or another, will be accessed by other users in the network.)

If meaningful group definitions will not be used for a workstation, then two of the IBM-defined AIX groups should be used. These are the *system* and the *staff* groups. Users (including yourself in your normal user mode) will be in the *staff* group. User *root* and yourself (in your administrator role) are in the *system* group.

3.6 Standard Userids

As distributed, AIX has userids and groups that are needed by the system. Do not alter these users and groups unless you are very certain about what you are doing. **Never login to any of these userids** (except *root*).

The user ids that are supplied with the system (in the form in which they appear in ***/etc/passwd***) are listed below. These are used for various purposes, such as file ownership and NFS functions. All these except *root* have been disabled for login in the distributed system. (They are disabled by *password = ** in ***/etc/security/passwd***.) The supplied userids are:

```
root:!:0:0:/:/bin/ksh
daemon:!:1:1::/etc:
bin:!:2:2::/bin:
sys:!:3:3::/usr/sys:
adm:!:4:4::/usr/adm:
uucp:!:5:5::/usr/spool/uucppublic:/usr/lib/uucp/uucico
guest:!:100:100::/usr/guest:
nobody:!:4294967294::4294967294::/:
lpd:!:104:9::/:
```

New users that you add will default into the *staff* group, unless you change the default. A workstation with only a few defined users will probably allow new users to default to the *staff* group. A larger multiuser system should have locally defined groups. You may find the *system* group useful if you have multiple administrators, since this group allows execution of many administrative functions. Other than these two groups (*staff* and *system*) the predefined AIX groups are for special system purposes. **Do not add users to the other**

[30] Some traditional UNIX systems permitted direct login using a group name. Among other problems, this reduced individual accountability of users, and made reconstruction of error situations more difficult.

predefined groups unless you have a special purpose for doing so. The groupids that come with the system are (in the form in which they appear in ***/etc/group***):

```
system:!:0:root
staff:!:1:
bin:!:2:root,bin
sys:!:3:root,bin,sys
adm:!:4:bin,adm
uucp:!:5:uucp
mail:!:6:
security:!:7:root
cron:!:8:root
printq:!:9:lpd
audit:!:10:root
ecs:!:28:
nobody:!:4294967294:nobody
usr:!:100:guest
```

We strongly advise you to not assign users to any existing group (except *staff*, of course) unless you are certain of the consequences. Some of these groups (such as *system, bin, security, cron*) are the group-owners of critical files and directories. A user in any of these groups, with a little effort, can subvert other security controls in the system.

3.7 Files Associated With User Accounts

The files that are associated with user administration are listed here with brief comments. Almost all files directly associated with user and group administration are in the ***/etc/security*** directory.

- ***/etc/security/.ids*** Do not edit this file. It contains the sequence numbers the **mkuser** command uses so that a new group or user always gets a unique *uid*/*gid*. The file is updated automatically by various commands invoked (internally) by **smit**. An example of this file is:

  ```
  6 221 12 206
  ```

  where:

  – 6 = next administrative *uid* number
  – 221 = next *uid* number
  – 12 = next administrative *gid* number
  – 203 = next *gid* number

- ***/etc/group*** contains basic group definitions. You would normally update this file through **smit**, but you may edit it directly. (A + (plus sign) beside an entry means to refer to the NIS server for additional entries. Never use the "+" unless you are certain NIS is installed and available in your network.)

- ***/etc/security/group*** contains additional group information, such as *adms* and *admin* flags. You would normally update this file through **smit**, but you may edit it directly.

- ***/etc/security/login.cfg*** contains stanzas for a variety of system-wide controls. There are no per-user stanzas. The controls are generally related to terminal and port usage. Some of the parameters can be set through **smit** with the "Change / Show Login Attributes for a Port" menu, while others can be changed only by directly editing this file. Comments in the file describe the functions and formats very clearly. Stanzas include:

- A group of parameters related to invalid logins. These parameters can delay or prohibit (for a period or indefinitely) additional logins after a failed login. These parameters can provide valuable protection for a system under attack by someone attempting to guess userids and/or passwords. If you have dial-in ports or are exposed to a large group of potential intruders, you should edit and use these parameters.
- sak_enabled - controls the availability of the secure attention function for a port. This is discussed later under the "Trusted Computing Base."
- auth_method (not defined in default file shipped with AIX) - defines different or additional authentication methods. See 3.7.1, "Additional Authentication Methods" on page 33 for discussion.
- herald parameters (the initial screen display before a user logs in) are in this file. You may edit and redesign the herald display. Be certain your herald contains enough new-line characters to clear the screen.
- usw - this is used only by the **chsh** command, and is a list of valid shells. Specify full path names. (The **chsh** command changes the initial program parameter in the user's line in ***/etc/passwd***. This command is normally generated by **smit**, and is not normally used directly.)
- maxlogins - sets the maximum number of direct terminal users who may be logged in at one time. (This parameter should be changed with the **chlicense** command, used in accordance with your AIX usage license.)
- logintimeout - the time within a login must complete.

See 3.9, "Other Topics" on page 35 for the **smit** menu associated with some of these controls.

- ***/etc/passwd*** contains basic user definitions. An "!" (exclamation point) in the password position is normal and causes the system to look in ***/etc/security/passwd*** where the encrypted password is kept. If the ! is replaced with an * (asterisk), the userid is locked. The **passwd** command replaces the * with an ! while defining a password. (A plus sign (+) beside an entry indicates a switch to NIS for additional entries.) You would normally update this file through **smit**, but you may edit it directly.

- ***/etc/security/passwd*** contains encrypted passwords, a time-stamp of the last update, and a flag indicating whether the password was updated by the administrator. (If so, the user will be prompted to change the password the next time he logs in.) You normally update this file through **smit**. You would edit it directly only in special circumstances, since you cannot directly enter the encrypted password.

- ***/etc/passwd.dir*** and ***/etc/passwd.pag*** are created by the **mkpasswd** command and contain small database structures to speed access to the userid administration files. Do not edit these files.

- ***/etc/security/user*** contains most of the user control parameters described in 3.3.1, "User Parameters in Smit" on page 17 and 3.3.2, "System Defaults" You would normally update this file through **smit**, but you may edit it directly. You should browse this file and become familiar with its format and contents.

- ***/etc/security/environ*** can contain environmental attributes for users. You can specify exceptions from the default user environment defined in ***/etc/environment***; for example, give a user a different *NLSPATH* (displaying messages in another language). You may edit this file directly, although we have not found a concise definition of exactly what (and in what format) can be placed in this file. There is one stanza for each defined user, but no control parameters are placed here by AIX.

- ***/etc/security/limits*** contains resource parameters. These can be important on multiuser systems to prevent a single user from consuming too much of the system's resources. There is one stanza per user and a default stanza. AIX will recognize all the following parameters, but some appear to have no effect in this release of the system. All of these parameters can be set through the *user* functions of **smit**. You can edit this file directly, but we recommend using **smit** instead.

  – *fsize* is the largest file a user can create. The default is 2,097,151 blocks. This is approximately 1GB of disk space. **We recommend a smaller value in a multiuser environment.** For example, 20,000 (which allows file creation up to 10 MB) might be a reasonable value. The smallest number that can be set through **smit** appears to be 8192.
  – *core* is the largest core file allowed, in units of 512 bytes.
  – *CPU* is the maximum number of CPU-seconds a process is allowed before being killed.
  – *data* is the largest data segment allowed, in units of 512 bytes.
  – *stack* is the maximum stack size a process is allowed, in units of 512 bytes.
  – *rss* is the maximum real memory size a process can acquire, in units of 512 bytes.

- ***/usr/lib/security/mkuser.default*** contains a few defaults used when creating a new user. You may edit this file directly. It contains the default group, the default initial program (shell), and the default home directory name for a new user.

- ***/etc/security/failedlogin*** contains an entry for every time a login fails. The file can be displayed with the **who** command:

  ```
  who -a /etc/security/failedlogin >> /tmp/check
  ```

  This example will redirect the output to a file called ***/tmp/check***.

  Do not edit this file. However, after an extended period you might want to delete it and allow the system to recreate it to recover disk space. This file is not as useful as it could be because it does not record invalid userids; that is, any userid that is not in your ***/etc/passwd*** file. Invalid userids are recorded as UNKNOWN rather than as the actual id entered. (Recording invalid userids is, itself, a potential security exposure, because entering a password when the system wants a userid is a common error.)

- ***/etc/security/lastlog*** has one stanza per user and contains information about several last logins (valid and invalid). Information from this file is displayed at user login time. You may display the file, but do not edit it. (The timestamps in the file are unreadable by humans, so that displaying it is of little value.)

- ***/etc/security/.profile*** is the prototype for the ***$HOME/.profile*** file for new users. You may edit this file and change it as required. It has no effect except when a new user is created.

- ***/etc/profile*** This file provides a system-wide login profile for all users. Any user's individual ***.profile*** (in his home directory) can override parameters in ***/etc/profile***. You may edit this file directly. Typical contents include:

  .

  – *TMOUT/TIMEOUT* defines the time (in seconds) that a user can be idle before he is automatically logged out of the system. *TMOUT* is used by **ksh** and *TIMEOUT* by **bsh**.

– Local options are often included here. Examples are local PATH variables for product libraries, a call to **/usr/games/fortune**, and so forth.

Several of these files have a second, older, copy in the /etc/security directory. The "old" copy has the letter "o" as the first character of the file name. For example, ***/etc/security/limits*** and ***/etc/security/olimits*** both exist. The **smit** processes (that is, the lower-level commands called by **smit**) copy the current file to the "old" version for recovery purposes. This is process is automatic. You should never touch the "old" files unless a disaster corrupts the operational files. The "old" files are normally one level down from the operational files; that is, the most recent change is not reflected in the old files.

3.7.1 Additional Authentication Methods

It is possible to have more than one method of authentication, for example: multiple passwords, a fingerprint scanner, one-time password response units, and so forth. AIX provides a simple interface for you (the system administrator) to specify additional authentication programs. You must provide the programming to perform the additional authentication. Appendix B, "Additional Authentication" on page 111 provides more detail and an example of an additional authentication program. We recommend using additional authentication only if there is a specific need for it.

The easiest additional authentication method to add is "two person authentication." This requires two passwords for logging into the system. The required setup is discussed in Appendix B, "Additional Authentication."

Note that the AIX terminology is unusual. You may have multiple *primary* authentication methods and multiple *secondary* authentication methods. Primary methods can reject a user; secondary methods cannot reject a user. More common terminology would refer to anything additional to standard password checking as "secondary authentication." This differing terminology can be confusing when talking with people who are serious about secondary authentication (in the more common sense of the term).

3.8 Verifying the User Environment

Security implementation requires both definition and maintenance. The security of a system is measured by how well the security state is maintained, as well as by how well it was originally defined.

Several "check" commands (**grpck, usrck, pwdch, sysck, tcbck**) and "list" commands (**lsuser** and **lsgroup**) are available for use by *root* (or anyone in the security group.[31] These commands can help you maintain your security environment.

[31] In early releases of AIX these check commands were automatically executed as part of the **sysck** command. This is not done in current releases of AIX.

The grpck Command

The **grpck** command verifies that all users listed as group members are defined as users, that the *gid* is unique, and that the group name is correctly formed. Other minor checks are also done. The **-t** flag causes the command to report errors and ask you for permission to fix them:

```
grpck -t ALL
```

This checks the group environment, and, if you answer *yes* to a prompt, it will erase the userids that do not exist or where stanzas in ***/etc/security/user*** have conflicting data.

The usrck Command

The **usrck** command verifies many parameters of a userid definition. The **-t** flag causes the command to report errors and ask for permission to take a standard fix. In some cases it will disable a userid by adding an expired expiration date to the user definition. The user's data is not affected. The user can be enabled again by removing the expiration date (using **smit** or directly editing ***/etc/security/user***).

Use this syntax to report problems and ask if they should be corrected:

```
usrck -t ALL
```

Never try to correct *root* using this command. If you want to try it, please read 3.9.1, "Repairing the root Userid" on page 36 first.

The pwdck Command

The **pwdck** command checks authentication stanzas in ***/etc/passwd*** and ***/etc/security/passwd***. If anything is wrong the standard fix is to remove the stanza or create a ***/etc/security/passwd*** stanza with an * (asterisk) in the password field.

This syntax will report problems and ask if they should be fixed:

```
pwdck -t ALL
```

We found that **pwdck** did not check for our specified password rules, such as *minalpha, minother, and lastupdate*.

The lsgroup and lsuser Commands

These commands are used internally by **smit**, but you can also use them directly. Direct use may be more convenient when you want to place their output in a file. The commands are:

```
lsgroup -f ALL >> /tmp/check
lsuser  -f ALL >> /tmp/check
```

In the form shown here, these commands create the file ***/tmp/check*** and write their output into it. There is too much output for direct display on the screen, so the output would normally be directed to a file. These commands display most of the control information about users and groups. These commands may be used by any user, but much more information is displayed when they are used by *root* (or any member of the *security* group).

The **lsuser** command is directly useful when used by *root* for a specific user:

```
lsuser joe
```

This command will display several lines containing control information for user *joe*. When used with the ALL operand, information is displayed for all users in

the system. Several formatting options are available. You could write local programs to extract and display locally-important information obtained from these commands.

The tcbck Command

This command is described in detail in Chapter 7, “Trusted Computing Base” on page 77. It requires on-going maintenance of a special database, and a certain amount of planning.

3.9 Other Topics

The command **mkpasswd /etc/passwd** can be used to create two small “hash” files to speed system lookup of entries in ***/etc/passwd***. If you have a larger system, with perhaps more than 100 users defined, this command may improve system performance slightly. You should run the command once. It will create ***/etc/passwd.dir*** and ***/etc/passwd.pag***. Once they are created, AIX will automatically update the hash files when **smit** is used to alter ***/etc/passwd***.

Security controls can be assigned to specific ports (such as /dev/ttyp0). These controls are stored in ***/etc/security/login.cfg***, which is discussed in 3.7, “Files Associated With User Accounts” on page 30. The **smit** menu is:

```
smit
  System Management
    Security & Users
      Login Controls
        Change / Show Login Attributes for a Port
          *Port NAME                        [/dev/ttyp0]
           Allowed LOGIN TIMES              []
           Login RETRY DELAY                []
           Number of FAILED LOGINS before   []
             port is locked
           INTERVAL for counting failed logins []
           REENABLE DELAY for locked port   []
           Is this PORT LOCKED?             []
```

The exact format and meaning of these parameters is documented in the comments in ***/etc/security/login.cfg***. The combination of the RETRY DELAY, FAILED LOGINS, INTERVAL, and REENABLE DELAY can provide good protection against repeated attacks on a dial-up port, in which the attacker is attempting to guess a userid or password.

The ***/etc/security/login.cfg*** file contains a default stanza and, potentially, a stanza for each port. We suggest using the default stanza rather than specifying values for each port (unless you need different values for different ports, of course). Unlike system user defaults (see 3.3.2, “System Defaults” on page 20), the port default values may be set using **smit** by entering “default” as the port name.

The files ***/etc/environment*** and ***/etc/profile*** contain similar types of profile and environment information. Both files are executed for every user at login time. The ***/etc/security/environ*** file can also contain similar environmental commands, but there is a stanza in this file for each user. When a user's login is complete, his initial environmental and shell variables are from (in order):

1. ***/etc/environment***
2. ***/etc/profile***

3. ***/etc/security/environ***
4. ***$HOME/.profile***
5. A shell configuration file specified by the *ENV* environmental variable (if used), or ***$HOME/.cshrc*** (if it exists and the C shell is used).
6. ***$HOME/.Xdefaults***, if it exists and AIXWindows is used.

The first three files in this list are system files and must be protected. Improper environmental or shell variables can create many security holes. The last three items are owned by the user, and the user can alter them in any way he wishes. Only the owner should have *write* access to these files, and there is no real reason for anyone else to have *read* access.

When assisting a user, try not to **su** to *root* from his session. If you do this, you are using his environment (with his *PATH*), and this opens a large number of exposures. If you must do this, then use full path names for all commands you use while executing as *root*.

Beware of a user who changes *IFS* (input field separator) in his profile. Do not allow it to be changed in ***/etc/profile***. A knowledgeable user can do clever things with IFS and cause endless trouble.

Beware of a user experimenting with ASCII terminal ″tricks.″ It is possible to send control strings to many ASCII terminals to set up various function keys. A clever user can send, for example, a series of commands ″hidden″ with the normal operation of a function key. In the proper circumstances, such as when *root* uses the terminal, these ″hidden″ commands can be used to cause commands to be executed under his UID without his knowledge.

The **who am i** command displays the login name associated with your terminal. This is unchanged by usage of the **su** command. The command **whoami** displays the current (effective) userid and changes when **su** is used. You normally want to use **whoami** and not **who am i**.

Never never never place the current directory in the *PATH* for *root*. If you log into the system as *root*, AIX will automatically remove the current directory (if it is specified) from the initial PATH. You can (but should not) restore it to the PATH after the login is complete.

3.9.1 Repairing the root Userid

If something goes wrong with *root*, you have a serious problem. Forgetting the *root* password is a common problem. New system administrators, while experimenting with various security options, may make the system so secure that no one can use it. To ″break into″ the system in order to repair it:

1. Find your system installation tape. (A CD-ROM can also be used.)
2. If possible, bring the system down by using **shutdown -F**. (Only *root* can use this command.)
3. Insert the tape in the tape drive, turn the key to the SERVICE position, and then press the yellow reset button. (You may need a power off/on cycle instead.)
4. Press F1 and Enter at the appropriate prompt.
5. Select ″Start Maintenance Mode for System Recovery″ from the displayed menu.

6. Select ″Access a Root Volume Group″.
7. Select ″Continue″ and respond to any additional prompts.
8. Select ″Access the Volume Group and start a shell″. When this completes you have normal access to all your system files, and you are executing as *root*.
9. You may need to set TERM=lft and EXPORT TERM in order to use your display in full-screen mode.
10. Use **smit** or other commands to repair your system.
11. Do a **sync** to ensure that the disk has been updated.
12. Perform **shutdown -F** when you have finished.
13. Return the key to the NORMAL position.
14. Re-boot the system from the hard disk.

Remember to give *root* a password when you are up and running again under your own system. Please note that this method of ″breaking into″ the system requires (1) physical access to the key for the RISC System/6000 console, and (2) a ″boot tape″ that is distributed with the AIX software.

3.9.2 Password Cracker Programs

Password *cracker* programs are available from a variety of sources (but not directly from IBM). These programs read encrypted passwords from ***/etc/passwd*** files and attempt to guess passwords that, when encrypted, will match a password in the file. Typically, the user provides a list of trial words (a “dictionary”), and the cracker program will try each word (and a number of permutations of each word) in the list.

In normal use, AIX does not maintain encrypted passwords in ***/etc/passwd***; rather, they are maintained in ***/etc/security/passwd***, in a different line format. Nevertheless, most *cracker* programs can be readily modified (if the source code is available) to work with ***/etc/security/passwd***. The user must have *read* access to this file, of course.

If you (an administrator) have a *cracker* program available, and have some time to work with it, we recommend using it. It cannot harm your system (assuming you have a trustworthy program), and may find any number of poor passwords set by your users. However, we do not recommend granting *read* access to ***/etc/security/passwd*** to any person who is not an authorized administrator.

The fact that AIX has moved the encrypted passwords where they are not easily available to *cracker* programs does not mean that you should not attempt to enforce good password quality.

Chapter 4. AIX File Security

Other than the login process, file security is the most apparent element of security to most AIX users. The discussion in this chapter considers only *local* files, files that are not accessed through a LAN or remote connection. The basic elements controlling file security are:

- The permission bits associated with the file
- The permission bits associated with the directory containing the file name
- The permission bits in all the directories in the file′s path
- Extended access control list parameters, if any
- The owner of the file
- The group-owner of the file
- The owner and group-owner of the file′s directory
- The owners and group-owners of all higher-level directories in the file′s path
- Programs executing with the effective userid of *root*

The individual elements are not complex, but the effect of various combinations can be confusing. This chapter discusses each of the listed elements.

4.1 File Systems

UNIX documentation discusses "files" and "file systems." This chapter is about files, but it is important to understand the terminology involved.

There are two common uses for the term "file system"in AIX. One is the total view or complete tree structure incorporating all the files from the top or root (/) directory, including all the other directories and files. This usage of the term is technically incorrect, but is widely used, nevertheless. "I don′t want any TCP/IP user to access my file system." "My file system never seems to have *fsck* problems." These two quotes informally refer to all the files on the owners′ systems. They are using "file system" to mean "all my files."

The technically correct meaning of "file system" is a contiguous space on a disk (or partition of a disk), or a logically contiguous disk area (managed by a logical volume manager) that contains all the controls and control blocks needed to manage its own internal space, including the allocation of files and directories. A normal UNIX file system has a super block, chains, and inodes as part of this control structure. In the case of JFS, one can equate "file system" with "logical volume."

AIX recognizes several types of file systems, and can have multiple instances of all of these in a given system:

- Journaled File System, the normal AIX File System.
- Network File System, for file sharing across networks.
- CD-ROM File System, for reading CD-ROM disks.
- Distributed File System (DFS), an optional component of the distributed computing environment (DCE).
- (Raw disk volume, not containing a standard file system. A number of database and other types of products use raw disks. The system may use

© Copyright IBM Corp. 1994

raw disk volumes for paging space, dump space, and other specialized uses.)
- (Diskettes do not normally contain file systems.)

A Network File System and a CD-ROM File System are special cases and are not discussed here. The Journaled File System is the standard AIX file system. The ″journal″ part of the name can be misleading since ″journal″ is often a database related term. AIX does ″journal″ (in the proper database sense) the changes to *inodes*.[32] That is, inodes changes are protected from corruption due to system failures. This is a substantial improvement over older UNIX systems that were very sensitive to file system (*inode*) damage.[33] The journal action does not apply to data in files. Data integrity in AIX is similar to that in other UNIX systems. (Of course, database products running under AIX may perform their own journal functions for their data.)

Generic UNIX documentation often refers to a *local file system*. For AIX, this is a journal file system (JFS). There are also references to a *virtual file system*(VFS). This is simply a coding and design technique that provides a uniform programming interface regardless of what type of actual file system is being used. JFS, DFS, NFS, and CD-ROM file systems are accessed through the VFS interface, providing a common API for the user. From the user and administrator point of view, VFS is all under-the-covers and requires no action.

The “journal” part of JFS works automatically. There is no user or administrator involvement. The **fsck** command has been modified to invoke JFS recovery checking. Booting AIX automatically runs **fsck** (including the JFS recovery functions), and no other actions are normally required.

A basic AIX system has several logical volumes and file systems:

```
  LV     FS          VFS type    Mount point or use

 hd1    /dev/hd1      JFS       /home
 hd2    /dev/hd2      JFS       /usr
 hd3    /dev/hd3      JFS       /tmp
 hd4    /dev/hd4      JFS       /
 hd5    /dev/hd5      ---       (boot)
 hd6    -------       ---       (paging)
 hd7    /dev/hd7      ---       (sysdump)
 hd8    -------       ---       (jfslog)
 hd9var /dev/hd9var   JFS       /var
 hd10   /dev/hd10     JFS       /usr/sys/inst.images
```

Notice that a separate logical volume (hd8 in this example) is used by JFS as the journal area. The special logical volumes and the logical volume manager itself are normally manipulated through **smit**, and this requires *root* authority. Other than protecting the *root* password, there is no routine security administrative work involved with the special logical volumes.

[32] An inode is a control block in a file system that keeps track of various pointers to files and free space.

[33] The well-known **fsck** command was frequently needed to repair this damage.

The mount Command

A file system must be "mounted" before it can be used. File systems can be mounted automatically upon startup or as needed by using the **mount** and **umount** commands. In general, *root* authority is needed to mount or unmount file systems. Either **smit** or the **mount/umount** commands may be used.

The **mount** command is intended to have three security levels, depending on the authority of the user invoking it. Any user can execute the command with no arguments. This will return the list of mounted file systems (JFS, NFS, and DFS). Members of the *system* group can use the command (with appropriate arguments) to mount a file system described in ***/etc/filesystems***. The file system must be currently unmounted, and the mount parameters cannot be altered from what is specified in ***/etc/filesystems***. Lastly, *root* can do almost anything with **mount**, mounting any file system over any directory, and overriding any parameters in ***/etc/filesystems***.

In addition, any user can mount directories over other directories to which he has write access. There is no direct security violation involved in doing this, but the results can be extremely confusing for the user (and perhaps for others accessing his files). Depending on your environment, you might consider (and experiment with) removing the **mount** from "world" access. Again, this is not a security consideration, but might be a usability consideration. (Later releases of AIX may remove the capability for a normal user to **mount** anything.)

CD-ROMs present, potentially, a serious security exposure. Files can be executed from a CD-ROM, including *suid root* files. Devices for *writing* CD-ROMs are becoming commonly available. Someone could create a dangerous *suid root* program on an external system and create a CD-ROM containing this file. If mounted on your system, the dangerous program will execute with *suid root* authority. You cannot control the external creation of CD-ROMs. You can control how CD-ROMs are mounted on your system. When you **mount** a CD-ROM, you can specify *-o nosuid*. (This is documented in the man page for **mount**.) You should use this when appropriate; you should probably include this option in any batch files you create to mount CD-ROMs.

4.1.1 Private File Systems

Adding a private file system involves creating a new logical volume, and then creating a file system within this logical volume. This is usually done through **smit**.

Simple workstations, using basic AIX facilities, may not need any private file systems. A user can store his private data in his home directory (which is normally ***/u/userid***) and in additional directories he creates below his home directories. **We recommend that you discourage the use of private file systems on workstations unless there is a particular need for them.**

Major software products, such as a database package or an office systems package, often reside in private file systems.[34] If such products are installed with the workstation, private file systems may be required.

Servers, typically with several application products installed (such as database managers), often have many file systems in addition to the basic AIX file

[34] The install process for these products may create the file system.

systems. The administrator (working as *root*) must create whatever additional file systems are required.[35] In general, the administrator will elect (through **smit**) to have the additional file systems automatically mounted whenever the system is booted.

The administrator must be concerned with the security controls of the new file systems. The ownership and permissions of the mount-point directory are important since all the contents of the file system will be under this directory. The file security management discussions in this chapter apply to locally-created file systems, as well as to system-created file systems.

If you create additional file systems, we recommend that the directory mount points have permission bits of -rwx------- (octal 700). Understand that you mount "over" the directory mount point, so any user files or directories below the mount directory are overlayed by the newly mounted file system and will not be available till you unmount the file system again. Installations with extreme security requirements can use a portable file system, that is unmounted and disconnected from the computer when not needed.

Removable file systems (which are usually "private") file systems have a unique exposure. When mounted, they function as normal file systems, including *suid* (and especially *suid root*) functions. A user could take his portable file system somewhere else and add many *suid root* programs. When mounted on your system, all these *suid root* programs could be disasterous. You have some control over mounting private file systems, since *root* or *system* (in some cases) authority is needed to **mount**. One of the **mount** options is *nosuid*. We recommend you always use this option when mounting portable file systems (including CD-ROMs), and (possibly) when mounting any private file system.

AIX does not support Journaled File Systems (or any other easily used file system) on diskettes.[36] AIX can read and write DOS-formatted diskettes with the **dosread** and **doswrite** commands. (These commands may not be included in the smallest *client* version of AIX 4.) Diskettes in **tar** format can also be used.

4.1.2 Inodes and Links

The normal UNIX file systems (including AIX's JFS) use a level of indirect file control that is usually hidden from the user. The administrator must understand some of the basic elements of this, since they are important for file security.

UNIX file access is usually like this:

```
directory entry --> inode  -->  data blocks
```

That is, the directory entry for a file does not point to the data for the file. It points to an *inode* that, in turn, points to the data.[37]

The security permission bits are attached to the inode, not the directory entry. Also, multiple directory entries may point to the same inode. A directory entry

---

[35] This may require a certain amount of disk space planning. Some understanding of LVM is required. Disk space planning is not discussed here.

[36] Solely from a security point of view, this is good. Diskette-mounted file systems, containing *suid* programs, can form a major security gap. (Predefined file systems in ***/etc/filesystem*** can specify a *nosuid* option to avoid this exposure if the predefined definition is used.

[37] This discussion ignores the details of inode data and indirect addressing through inodes. These details are not relevant to routine security processes.

contains a “name” for a file, such as /u/trial/data. An inode has an identification number, but no file “name.” (A number of low-level UNIX commands exist to manipulate inodes directly. These commands should not be used by normal users, although they do not bypass any security functions.)

A more general picture might be:

```
/u/trial/data       -->
/xyz/j/g34/check    -->    inode 317      -->  data blocks
/joes/stuff         -->
```

In this example, a single file (based on inode 317 within some file system) has three directory “links.” The same file has three very different “names.” Permission bits (and the UID and GID) are stored in the inode. Accessing the file through any of the names will provide the same permissions and owner controls. These extra names are provided by *symbolic links* or *hard links*. (A symbolic link can function across file systems and is not deleted if the target inode is deleted. A hard link works only within a given file system and can be a controlling element in deleting the inode and file data.)

Similar links can exist within directory levels. For example, the /xxx directory could be linked to the /etc directory. This means that file /xxx/my/data is really /etc/my/data. The base AIX system does some of this by default. For example, there is no ***/u*** file system. Instead, /u is linked to ***/home***. File /home/her/data can also be accessed as /u/her/data. The same *file* is accessed in both cases, although different directory structures are used. The security implications of directory links are discussed later.

4.1.3 Ownership

Every file (including directories) has an owner and a group. The owner and group identifiers (the UID and GID) are stored in the inode. The owner is, initially, the user who created the file. The group is the current group of the owner when he created the file.[38] *Root* can change the owner of a file by using the **chown** command, and can change the group owner with the **chgrp** command. In AIX, normal users cannot use the **chown** or **chgrp** commands, because these functions can indirectly lead to security exposures. In some versions of UNIX, these two commands are available to normal users.

The UID and GID owners of a file are set when the file is created and are only changed by explicit change commands. Note that ownership is by UID (owner) and GID (group owner), not by userid and groupid. If a userid (and his corresponding UID) is deleted from the system, files owned by that UID remain in the system. Removing a userid does not remove his files. However, there is no longer a userid associated with the UID, and the files are said to be “unowned.” If another userid is later assigned the same UID, he then owns all the files associated with that UID.

If you are concerned about this, you might *lock* a user account instead of removing it. In this way his files are still “owned.” Files owned by a user can be located with the **find** command, although, if NFS or portable disk drives are involved, this becomes complex.

[38] Another option for assigning group ownership is discussed later.

The group or group-owner is the GID of a group defined in ***/etc/group***. Any member of this group (as defined in ***/etc/group***) has whatever rights a group-owner has for the file.

You should understand how the ownership of a file is affected by the **mv** and **cp** (move and copy) commands.

- The **cp** command always creates a new file, and the user of the command becomes the owner of the new file. The user must have sufficient permissions to read the source files. (This requires at least *execute* permission for all the directories in the path of the input file, and *read* permission for the file itself.) He must have *write* permission for the target directory.
- If the **mv** command is used to move a file within a file system, the ownership of the file is not changed. The user of the **mv** command must have *write* permission in the target directory, and sufficient permissions to read the file.
- If the **mv** command is used to move a file to another file system, a new file is created in that file system, and the current user is the owner of the file. The user must have *write* permission in both the source directory (to delete the file) and the target directory (to create a new file). He must also have appropriate read permissions for the source.

4.1.4 Permission Bits (Basic)

Files (and directories) have *permission bits*. An administrator must thoroughly understand these. These are sometimes referenced as "mode" bits, but we will use the term "permission bits" or "permissions." The basic permission bits are quite simple.

There are 12 permission bits:

- Three *system* bits (which are not directly displayed),
- Three *owner* bits that describe what the owner is permitted to do,
- Three *group* bits that describe what any other user who is a member of the group may do, and
- Three *other* or *world* bits that describe what any other user can do.

Permission bits are often displayed as nine bits. (The three high-order system bits are displayed in special ways.) A typical permissions display is:

```
rwxr-xr--
```

The first three characters displayed are the owner bits, the next three are the group bits, and the last three are the other bits. Within each three-bit group, the first bit is for *read* permission, the second for *write* permission, and the last for *execute* or *search* permission.

By convention, a letter means the bit is on, and a dash means it is off. In the example above, the owner has read/write/execute permission, anyone in the file's group has read/execute permission, and everyone else has read permission. Permissions are not hierarchical; *write* does not include *read*, and *execute* does not include *read*.

Permission bits are often written in octal. The above example, in octal, is 754. Octal is convenient because the first digit represents the owner's permissions, the second digit is the group's permissions, and the third digit is everyone else's

permissions. When using octal notation, sometimes four digits are shown. In this case, the first digit contains the system permissions (which are explained later).

The *execute* permission is not as simple as it might appear:

- For binary programs (produced by a compiler, and linked for execution) it functions in the obvious way.
- Shell scripts are only partly limited by *execute* permissions. If the name of a shell script is entered on the command line in the normal manner, the *execute* permissions are examined. If the shell script is named after a “dot” command, or named when starting a new shell, the *execute* permissions are not examined. In these cases, the *read* permissions are examined instead. The logic, such as it is, is that the current shell is reading the shell script as a data file; the AIX kernel is not executing the shell script.[39]
- The meaning of *execute* permission for directories is described in several sections below.

4.2 Basic File Security Concepts

The basic elements of AIX file security[40] are quite simple. The permission bits for the file and the permission bits for the directory containing the file are the key elements. In UNIX, a directory is a type of file. As a file, it has permission bits in its own inode. **Permissions to read and write in a directory are independent of permissions to read and write the files named within the directory.**

This is a critical concept, and is not intuitive to anyone whose background includes DOS, OS/2, or MVS.[41] A user with *write* permission for a directory can create, rename, or remove a file. Directory *execute* permission is required to access the directory (when looking for a file, for example).

A file′s permission bits are involved when opening, reading, writing, updating, or executing the file. Directory permission is required to *find* a file before opening it for use.

The *owner* of a file or directory can always change the permission bits. The owner is usually the user who created the file or directory, but ownership can be transferred (by *root*) to another user. Three permission bits apply to the owner; the owner can set these to protect himself against his own mistakes. For example, the owner of a file can set the owner permission bits to “r-x.” This will prevent the owner from writing in the file. However, the owner can always change the permission bits for the file (using the **chmod** command) if he really wants to write in it.

Directory permissions are especially important for AIX system files, such as those in ***/usr***. You should not allow users to add files to ***/usr*** and its subdirectories (unless there is a good reason for doing so). You control this by

[39] This is standard UNIX behavior, and would not be practical to change for AIX.

[40] Unless otherwise qualified, ″file″ means an ″ordinary JFS file″ in the AIX sense.

[41] It is intuitive to those with AS/400 backgrounds, because the AS/400 separates the authority to control a file (create, delete, and so forth) from the authority to use the data in the file.

not allowing *write* permission for "others" in any of these directories.[42] This is the default condition when you receive AIX. You can list directory permissions with the command:

```
ls -ld dirname
```

where *dirname* is the pathname of the directory you wish to list. (You can also display directory permissions by listing (with the **ls -l** command) the contents of the directory which is one level above the directory whose permissions you want.)

A directory cannot be executed, and the "x" permission bit is used to control the ability to search the directory. To **cd** to a directory, or to use it as part of a path name, one must have *search* permission for the directory. To list the files in a directory, one must have *read* permission for the directory.

You (an administrator) must care for permission bits in directories because all other security depends on these. Consider the following extreme example:

1. I (a bad guy) find that the permission bits in the root directory are set to rwxrwxrwx. This means that I (a normal user) can write into the root directory.
2. I rename /etc to /trash, using the **mv** command. I can do this because I can write in the root directory.[43]
3. The system soon crashes or hangs because all the /etc files have disappeared. The administrator will need to boot from a maintenance CD-ROM or tape and spend time diagnosing and fixing the problem.

While this is not a very useful attack on a system, it illustrates the critical nature of directory permissions. A more sophisticated attack, in the same situation, might be to create a new directory and copy all the existing /etc contents into the new directory. I (the bad guy) am the owner of all these copied files. I then rename /etc to /trash and rename my newly-made directory to /etc. The system continues to function because all the /etc files are present, in their copied form. I (the bad guy) am now the owner of all the operational /etc files, and I can alter these as I please.[44]

This last type of attack can be used at any level of the directory tree. If a user can write into a directory, he can subvert all files and subdirectories in the directory. That is, any files and lower-level directories can be copied and manipulated. The copied versions can, in effect, become the "real" versions.

The administrator must ensure that system directories (at any level in the directory tree) are not writable by normal users. In general, no directory should be writable by the "world." The ***/tmp*** and various ***lost+found*** directories are special cases, discussed later. As distributed, AIX contains a number of world-writable directories. These can be listed with the command:

```
find / -perm -0007 -type d -print
```

---

[42] You must also be careful not to add users to any groups that have write permission for system directories or files.

[43] I cannot remove /etc because this implies removing the files and subdirectories contained in the directory and I do not have authority to do this. I can delete a simple file or an empty directory (by using **rm -R name**)

[44] In practice, this attack would need more steps. As I would be the owner of the new /etc files, programs requiring *suid root* would not function properly. This is not the appropriate document to describe the full details of this particular attack.

Some of these world-writable directories use the "sticky bit," explained later, as an additional control.

The use of directory permissions is so important that we will restate the basic principles again:

- To use a file (as data or an executable program) the user must have *search* (that is, *execute*) permission for *all* directories in the path. He must also have appropriate permission for the file itself, of course.
- To list a directory (with the **ls** command, for example), a user must have *read* permission for the directory.
- With write *permission* for a directory, a user can add new files and subdirectories, move (rename) files, and possibly delete files and subdirectories in the directory.[45] These actions can be taken regardless of the permissions on the files in the directory. In one way or another, a user can subvert any file, lower-level directory, or files in a lower-level directory if he has *write* permission to the directory.

As distributed by IBM, AIX has all file and directory permissions, owners, and group owners correctly set for secure operation. You should consider the security consequences before changing any of these controls.

4.2.1 The ls Command

The **ls** command is probably the single most important command for you as a security administrator. (The **find** command is the second most important.) You must understand the detailed information it displays. AIX also provides the **li** command that is very similar to **ls**. We suggest you concentrate on the **ls** command, since it is standard in all UNIX systems, and learn to use several of its optional flags.

The basic **ls** command displays a list of the files in the current directory and displays nothing else.

For more information, you will normally use one of the following forms:

```
ls -al
ls -ld
ls -l /some/file/name
ls -ld /some/directory/name
```

The first form, **ls -al**, displays information about all the files in the current directory, including "hidden" files (whose names begin with a period). The second form, **ls -ld**, displays information about the the current directory itself. The third form, **ls -l /some/file/name**, displays information about a particular file. The last form, **ls -ld /some/directory**, displays information about a specified directory.

The general format of the display is shown in Figure 1 on page 48. (The inode number, shown in the figure, is not displayed by the forms of **ls** shown above. The flag "i" can be used to display inode numbers, but these are not useful in most situations.)

---

[45] Deleting a subdirectory requires permissions to delete all the files in the subdirectory.

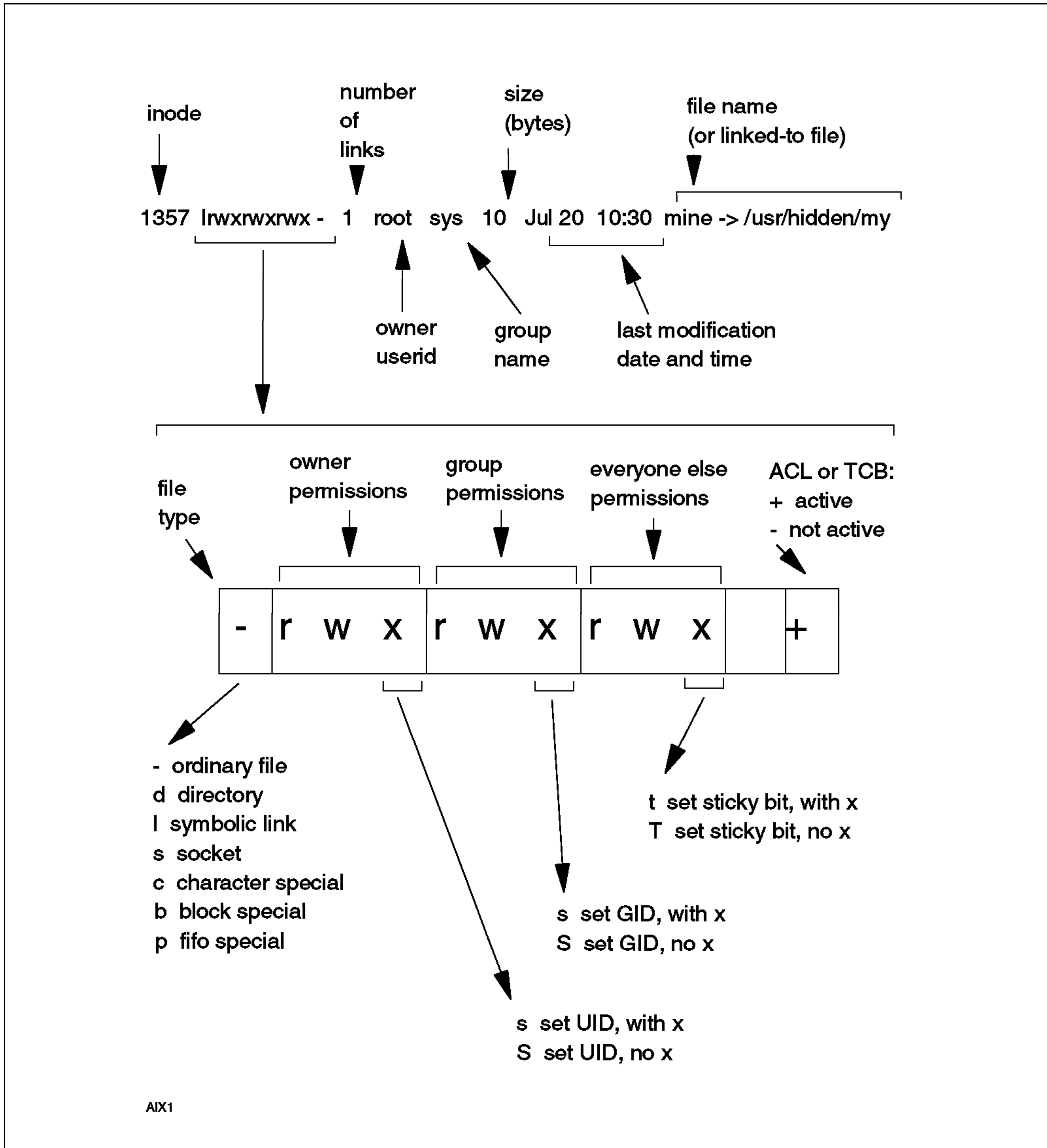

Figure 1. Interpreting Fields Reported by an "ls" Command

The *set UID, set GID,* and *sticky* bits (all shown in the figure) are described below.

The owner's userid is shown by all the long forms of **li**. Remember that a file (or directory) owner can change any of the attributes of the file except the *owner* and *group* names. Only *root* can change these. An inode contains the UID and GID of the owner, not the names. If the UID (or GID) is no longer registered in ***/etc/passwd*** (or ***/etc/group***), then a number (the UID or GID) is displayed instead of the name. This is an indication of an ownerless file.

A great deal of information relevant to security is packed into ***li*** output, and you should understand all of it.

4.2.2 Permission Bits (Advanced)

UNIX uses 12 permission bits. Of these, nine are the basic r/w/x permissions for owner/group/other, and were described previously. The three remaining bits are somewhat more complex. They are:

1. The set UID (or *suid*) bit
2. The set GID (or *sgid*) bit
3. The save-text (or *sticky*) bit

These bits are critical for security controls, and are displayed by modifying the normal "rwxrwxrwx" string used to display the basic permission bits. For display purposes, these bits modify the three "x" bits in the normal display.

The *suid* bit is displayed by changing the "x" in the owner "rwx" to an "s," and so forth. This is shown in Figure 1 on page 48.

The *suid* bit means that the program will run under the authority of the UID of the owner of the file. (Executable files are normally executed under the authority of the UID of the user who is logged into the system and asking to have the file executed.) For example:

```
-r-sr-xr-x 1  root  sys  3254  Jun 1 11:30 myprog
```

has the *suid* bit set. If I (logged into the system as user *joe*) execute **myprog**, it will execute with *root* authority. Since *root* can bypass almost all security controls, this could be dangerous.[46] In this case, **myprog** might be a copy of the Korn shell (or something similar). By executing **myprog** (with *suid* to *root*), I effectively become root. I can enter any system command using this shell, and all the commands will run under root′s authority. This situation, a shell with *suid* to *root*, is a prime goal of any system intruder.

These bits are set with the **chmod** command, using either symbolic operands or a 4-digit octal operand.

The *suid* bit can be set (using the **chmod** command) only by the owner of a file or by *root*. It is automatically removed by the **cp** (copy) command. There is no direct way for a normal user to create a *suid root* file.

The *suid* function can be used with owners other than *root*. It can be used, for example, to ensure that a file is accessed only by a certain program. For example:

```
-rw------- 1  joe   eng  5432  Jun 2 13:45 mydata
-r-sr-xr-x 1  joe   eng  2345  Jun 1 11:30 myprog
```

permits anyone to execute **myprog**. Only userid *joe* can access ***mydata***. Since anyone can execute **myprog**, and since **myprog** uses *suid* to execute as *joe*, anyone can access ***mydata*** only by executing **myprog**. The assumption is that

[46] The *suid root* programs supplied as part of AIX can be trusted to behave correctly. As an administrator, you must be very careful about installing new *suid root* programs that anyone can execute.

this program contains whatever security controls it needs to manage proper access to ***mydata***.[47]

A typical AIX system has a few hundred programs that *suid* to *root*. The administrator of a multiuser system should ensure that any additions (new programs that *suid root*) are known, trusted programs. AIX provides a utility, **tcbck**, that can help manage this. It is described in Chapter 7, “Trusted Computing Base” on page 77.

The “set group” (*sgid*) function works just like the *suid* function, using a file's group identity instead of the owner identity. The *sgid* bit has a special meaning when used with a directory, where it determines how group ownership for new files is assigned.

AIX ignores the *suid* and *sgid* bits when executing shell scripts. That is, only compiled “object code” programs can *suid* to another UID for execution. Some UNIX systems permit shell scripts to *suid*. In principle, this is useful. In practice, it is an endless source of security breaches and has been removed from AIX for this reason.

The *sticky* bit has been used for multiple purposes. In earlier systems it was used to indicate that a program should be retained in memory after execution, in order to improve system performance. This function is not used in modern UNIX systems. Instead, it is used with directories to further limit who may alter entries in the directory.

In a normal directory (without the *sticky* bit), any user with *write* access can move or remove files in the directory. This is a severe exposure with directories, such as ***/tmp***, that are world-writable. When the *sticky* bit is set, only the owner of a file can delete it, even if the directory is world-writable. (As might be expected, the directory owner and *root* can also delete files from the directory.)

Note that any security information effective in a specific directory is not propagated to lower directories; each directory obeys only its own permission bits.

Permissions (for files or directories) are not cumulative for the owner, group, world fields. In general, the owner field provides more permissions than the group field, and the group field more permissions than the world field, but this is not required. For example:

```
-r--rw-rwx 1  joe   xyz  3210  Jun 3 15:15 mystuff
```

has rather unusual (but valid) permissions. The owner (*joe*) cannot write or execute this file. (He can change the permissions, of course, and add more permissions for himself, but, as shown here, he cannot write or execute the file.) Any member of group *xyz* can read or write the file. Anyone else (other than the owner and any members of the group) can read, write, or execute the file.

Stated a different way, the permissions assigned to the owner, group, or world are also restrictions. The owner, for example, is not considered part of the

[47] You should completely understand this example. It incorporates many of the key elements of permission bits, and represents a practical way to control use of shared data.

“world.” This aspect can be used to exclude certain users from accessing a file. This can be done by creating a new group containing all the users to be excluded. The file′s group-owner is then changed to this new group name. The group permissions are set to ′---′. The result is that no member of the group can access the file, even if the file has full access for the rest of the world.

File permission bits are verified when a file is opened. The commands **mv** and **rm** do not open a file (unless the **mv** is to another file system). Thus it is possible to remove a file that you do not have permission to open, as long as you have write (w) permission for the directory containing the file.

This can be prevented by use of the sticky bit (SVTX). When the file is a directory, the sticky bit is used for additional security. When it is set, only the owner of the directory or the owner of a file in the directory can delete or rename the file, even if ″the world″ has write permission to the directory. This is useful for files in ***/tmp*** because this directory must have *write* permission for the “world.” Error logs, daily security or accounting reports, and similar files are usually written in ***/tmp***. This use of the sticky bit prevents just anyone from deleting such files. We recommend setting the sticky bit for ***/tmp***. However, please note that some major software packages may not work (or require special setup) in this case. The sticky bit is set with the **chmod** instruction.

Directory Permissions Summary

To summarize, permission bits used with directories have the following meanings:

- The *suid* bit is not used.
- The *sgid* bit is also named the *group inheritance flag* when used with a directory. It controls what group name (actually, what GID) is assigned to new files created in the directory (including new subdirectories). If this flag is set, the GID assigned to the directory itself is used as the GID for any new files created in the directory. If the flag is not set, the GID of a new file is the current group of the user who created the file. (A user can change his current group with the **newgrp** command.)

  The group inheritance function can be set as the default for a file system by defining the *grpid* parameter in the stanza for the file system in ***/etc/filesystems*** or when the file system is mounted.
- The *sticky* bit means that, even though the directory is writable by the current user, only the owner of a file in the directory can delete (“unlink”) a file. (The owner of the directory and *root* can also delete files.) When used this way, the bit is sometimes called the *link permission flag*. Note that this flag also prevents nonowners from renaming a file (with the **mv** command). This flag is commonly used for shared directories, such as ***/tmp***, and various spool and mail directories.
- *Read* permission (in owner, group, or world fields) permits a user to read the directory (but not the files within the directory). The **ls** command, among many others, reads directories. It will not list a directory unless the current user has *read* access to the directory.

  Read permission in a directory is required in order to use wildcards when referencing the directory.
- *Write* permission (in owner, group, or world fields) permits a user to add, delete, or change entries in the directory. A file can be added, deleted, or renamed. An existing file cannot be read or written (unless the user has

appropriate permission to the file), but it can be deleted or renamed since these actions take place in the directory, not the file itself. This is why directory *write* permissions are so important for security administration.

- *Execute* permission is called *search* permission when applied to a directory. It permits the directory to be used as part of an explicit path name. To access file ***/u/mydir/file3***, the caller must have *search* access to the root directory, the u directory, and the mydir directory. *Search* permission does not permit listing or reading the whole directory; it permits use of a single entry in the directory. The user must, by some external means, know the name of the entry (that is, the path name) of the file he wants.

4.2.3 The umask Variable

Every file (and directory) has permission bits. The owner can change them with the **chmod** command. The initial, default, permissions set when a file is created are controlled by an environmental variable named *umask*. For reasons going back to the early days of UNIX, the *umask* value is used in an odd way. Default permissions are established by assuming permissions ("rwxrwxrwx" or octal 777 for directories, or "rw-rw-rw-" or octal 666 for normal files) and removing the permission bits specified in the *umask* (which is always expressed in octal).

The default *umask* is 022 (octal). Therefore, default permissions are:

```
666 removing 022 = 644 = rw-r--r--      (for a file)
777 removing 022 = 755 = rwxr-xr-x      (for a directory)
```

The *umask* is an environmental variable that can be changed by the user with the **umask** command (which is a shell command). There is no way to enforce a standard value for users. The default values are suitable for most uses. A different default can be set by placing a **umask** command in a user's ***$HOME/.profile*** file, for example. However, the user can change the value at any time. A user's initial umask value can be set through **smit**.

You can check your default with the **umask** command (with no operand).

4.2.4 File Timestamps

UNIX systems, including AIX, maintain three timestamps for files (including directories). These can be important for resolving security questions. The timestamps are:

1. *atime*. This is the time the file was last accessed. In effect, this is the last time the file was opened.
2. *ctime*. This is the last time the *inode* for the file was changed. (It is not the *creation* time, unless file creation was the last event for the inode.) The inode is changed when permissions are changed, the owner is changed, the file size (number of clusters) is changed, and so forth.
3. *mtime*. This is the last time the contents of the file were changed. This generally means the file was opened for output. This time can easily be manipulated by *root* with the **touch** command. For example, the command **touch 0101000095 afile** will set the *mtime* to Jan 1 1995. (Only *write* authority to the inode is needed to manipulate the times, but the AIX **touch** command will not perform this function for normal users.)

The long forms of the **ls** command normally list the *mtime*. The -c flag can be used to list the *ctime* instead. The -u flag can be used to list the *atime*. The **find** command can reference all three timestamps.

4.3 The ACL Commands

AIX has an additional security function for files. This is the access control list (ACL) facility. This is not a standard part of "traditional" UNIX. Modern UNIX systems usually have an ACL-type function, but the commands and exact functionality differ between vendors.

AIX ACLs can provide much finer-grained access control than can be obtained with permission bits. As a general case, explicit ACL control is not normally used with workstations. It may be used within specific applications on servers. That is, normal AIX usage typically does not involve individual users randomly assigning ACLs as the mood strikes them. (Although such is possible and there are no controls to prevent it.) A typical usage would be a planned set of ACLs for the payroll department′s files, for example.

Every file (and directory) has a "base ACL" because the standard permission bits (old term) are also the base ACL (new term). The extended ACL functions (new term) are usually simply called the ACL functions.

Base Permissions

Base permissions are shown by ACL-related commands in the following format:

```
         attributes:  SUID, or SGID or SVTX in any combination
   base permissions:
        owner(name):  rw-
       group(group):  r-x
             others:  -wx
```

where:

SUID means *setuid*
SGID means *setgid*
SVTX means Savetext (sticky bit)

Extended Permissions

Extended permissions allow the owner to define access to a file more precisely. Extended permissions extend the base file permissions (owner, group, others) by permitting, denying, or specifying access modes for specific individuals, groups, or user and group combinations. Any user can create an extended ACL for a file he owns.

The permit, deny, and specify keywords are defined as follows:

permit grants the user or group the specified access.
deny restricts the user or group.
specify precisely defines the file access.

If a user is denied a particular access by either a deny or a specify keyword, no base permission or general extended permission can override that denial. When both a user and group are defined in an extended permission, only the specific user and group combination receives the access. There is an ″and″ relation between the elements in a list. The *enabled* keyword must be included in the access control information for the extended permissions to take effect. The default value is the *disabled* keyword. Using **chmod** with an octal operand is one way to set the *disabled* state.

Extended permissions are shown in the following format:

```
attributes:  SUID, or SGID or SVTX in any combination
base permissions:
       owner(frank):  rw-
      group(system):  r-x
             others:  ---
extended permissions:
      enabled
        permit  rw-  u:dhs
        deny    r--  u:chas, g:system
        specify r--  u:john, g:gateway, g:mail
        permit  rw-  g:account, g:finance
```

The first line of the extended permissions describes its status: enabled or disabled. If disabled, the extended ACL information has no effect; only the base permissions are effective. The second line explicitly grants *dhs* read (r) and write (w) permission on the file. The third line specifically denies *chas* read (r) access only when he is a member of the *system* group. The fourth line grants *john* read (r) access if he is a member of both the gateway and the mail groups. The fifth line permits read and write access to this file for users that belong to both the account and finance groups.

The meaning of an ACL can become complex for a user who is a member of multiple groups. An ACL might include entries for several of the user's groups, and these may conflict. For example, a user may belong to GROUP1 and GROUP2. A given ACL may provide *read* access for GROUP1 and *execute* access for GROUP2. These conflicts are resolved in this order:

1. If a SPECIFY operand exists for any of a user's groups (or for his own userid), the SPECIFY will set a maximum access level. If multiple SPECIFYs exist (for different groups and/or the userid), the least-common denominator of all the SPECIFYs is used. Access rights will never be higher than this, and may be less, due to DENY permissions.

2. All (positive) access permissions (for the user and all his groups) are added together.

3. All DENY (negative) access permissions (for the user and any of his groups) are then subtracted. The result is further limited by SPECIFY restrictions (if any).

A DENY function, in a sense, is more powerful than PERMIT functions because a single DENY can override any number of PERMITs. This result may surprise users and administrators, but it is a logical result of ACLs, DENYs, and many-group operation. If a user is unable to access a file, and you cannot understand why, you should check the ACL for any DENYs associated with groupids. The user may be a member of the DENYed group. The same effect can be caused by group-level SPECIFYs.

The ACL commands referenced here are primarily for extended ACL functions, but they can be used instead of **chmod** to control base permission bits also. The commands are:

- **aclget** gets the ACL for a file
- **aclput** sets the ACL for a file
- **acledit** combines **aclget** and **aclput**.

The **acledit** command lets the owner change the access control information for a file. This command displays the current access control information and lets the file owner change it (using the editor specified by the *EDITOR* environment

variable). Before making any changes permanent, the command asks if you want to proceed.

The *EDITOR* environment variable specification must specify the full path to the editor.[48] For example:

```
     EDITOR = /usr/bin/vi
or   EDITOR = /usr/bin/e
```

(If the **acledit** command is operating in a trusted path, the editor must have the trusted process attribute set. This is discussed in a later chapter.)

The **aclget** command displays the access control information of a file. (An *outfile* parameter can be used to send the displayed information to a file.) The information includes attributes, base permissions, and extended permissions. For example:

To display the access control information for the file ***status*** enter:

```
aclget status
```

To copy the access control information of the ***plans*** file to the status file (using a pipe), enter:

```
aclget plans | aclput status
```

The **aclput** command sets the access control information of a file. (The *-i* flag can be used to obtain input from a file rather than from system input. If the access control information in the file specified by the InFile parameter is not correct when you try to apply it to a file, an error message preceded by an asterisk is added to the input file.) For example:

To set the access control information for the status file with information stored in the ***acldefs*** file, enter:

```
aclput -i acldefs status
```

To set the access control information for the status file with an edited version of the access control information for the plans file, you must enter two commands.
First, enter:

```
aclget -o acl plans
```

This stores the access control information for the plans file in the ***acl*** file. Edit the information in the ***acl*** file, using your favorite editor. Then enter:

```
aclput -i acl status
```

Do not depend on extended ACLs in heterogeneous networks, since non-AIX systems will ignore them. Only AIX systems will observe extended ACLs over a network. For reasons of compatibility, we recommend that you discourage use of the ACL functions except for preplanned uses with major applications.

[48] This is partly to avoid security exposures due to improper PATH parameters.

The chmod Command

There are two methods for setting and controlling permission bits: the **chmod** command and the ACL set of commands. The access control list commands are primarily for working with extended access list functions. The **chmod** command is the primary tool for changing base permission bits.

chmod operands can be octal (sometimes called ″absolute″) or symbolic. Octal notation is common, and this is imbedded in many script programs and shown in most texts. A symbolic operand can do relative changes such as *add (+)* and *subtract (-)* or *clear & set (=)*. An octal operand simply replaces the total permissions value.

The use of an octal operand will disable the extended ACL parameters (if any) associated with the file. If you use extended ACLs, you must use **chmod** with symbolic operands when working the files containing the extended ACLs. (An alternative is to use the ACL editor.) For example you should use **chmod a+rw myfile** rather than **chmod 644 myfile**. This may be an unfamiliar requirement, and it is very difficult to remember not to use octal notation. It is almost possible to enforce the use of only symbolic operands. The **tcbck** command can locate files with disabled extended ACL′s. See 7.1, "TCB Description" on page 78 for an introduction to the **tcbck** command.

4.4 Files That Grow

AIX has some files that grow until the file system is filled or until you (the administrator) take action. Some of the growth is due to normal AIX operation and some is due to error situations. The **skulker** command (shell script, explained in 9.3, "The skulker Script" on page 92) can be configured to automatically remove many unneeded files. You must manage the others that **skulker** misses or does not inspect.

You may remove the following files whenever you encounter them (unless there is a particular need for the data, of course):

- Older ***smit.log*** files (in various directories) (You might want to keep the more recent logs. They sometimes help you remember exactly what you did a few weeks ago.)
- ***smit.script*** (in various directories)
- ***core*** (in any directory) (Unless you have someone who can read AIX core dumps!)
- /usr/spool/*/* (if more than a few days old)
- /usr/tmp/* (if more than a few days old)
- /tmp/*/* (if more than a few days old)
- $HOME/mbox (Should be managed by every user in his home directory. Sometimes a user must be reminded to manage his file if it is using too much disk space.)

The following administrative files are your (the administrator′s) concern. If they grow too large, you can edit them or remove them. The system will create a file again (starting with minimum disk space, of course) when it needs it.

- /var/adm/cron/log
- /var/adm/wtmp
- /var/adm/pacct
- /var/adm/messages
- /var/adm/acct/nite/*

- /var/adm/dtmp
- /var/adm/qacct
- /var/adm/acct/sum/*
- /var/adm/acct/fiscal/*
- /var/mail/*
- /etc/utmp
- /etc/security/failedlogin
- /var/adm/sulog
- /audit/trail

The INed file manager creates backup files. If you delete files or whole directories using INed, it will usually make backups in special hidden files before deleting the files or directory. You can devote a whole session to deleting files and directories and find you are using more disk space after the deletions than before. We recommend not using INed for file/directory deletion until you (and your users) understand its backup operations. These hidden files are usually placed in the user′s home directory, and their names begin with one, two, three, or four periods. If you use INed to delete these backup files/directories, it will sometimes make a ″deeper″ layer of backups; you should manage these files with the **rm** command. **skulker** deletes some of these backups but may not get all of them. You can erase the contents of a file (and free disk space) without deleting the file by copying ***/dev/null*** to the file.

4.5 AIX Version 4 Error Logging

Security exposures sometimes happen because of errors. AIX has a good error logging and reporting facility. **You should list the error log regularly.**

The **errpt** command is used directly, or with **smit** as follows:

```
smit
-Problem Determination
--Error Log
---Generate Error Report
   Change / Show Characteristics of Error Log
   Clean Error Log
```

The operating system records selected hardware and software failures in the system error log. This selection can be modified using the **errupdate** command. An error report can be obtained in summary or detailed form. **We recommended that error logging should always be active.** It is active as long as **errdemon** is running, and this is started automatically when the system is booted.

Using the **smit** menu:

```
smit
-System Environment
--Change / Show Characteristics of Operating System
```

you can limit the size of the error log.

4.6 Other Comments

AIX and other UNIX systems use *symbolic links* heavily. The **ls -l** command denotes these with an arrow in the name field and an "l" as the first character of the permissions field. For example:

```
lrwxrwxrwx   1  root   system   5 Jul 22 1993 u -> home
```

means that ***u*** is a symbolic link to ***home***. Note that it appears that everyone has *write* permission to ***u***. This is misleading. The permissions in a symbolic link have no meaning.[49] The effective permissions are taken from the target name. In the above example, anyone working with ***u*** must work under the permissions set by ***home***. (In this example, ***u*** and ***home*** are directory levels, but the same concept applies to both directories and files.)

UNIX (including AIX) has no simple way to detect when the target of a symbolic link has been deleted. Over time, symbolic links with missing targets may accumulate. These can cause errors that puzzle normal users. No direct security concerns are caused by this, but you should be aware of the problem. The most common effect is that a file appears to exist when accessed by one method, but appears to be missing when accessed a different way.

The environmental parameter *ulimit* controls the maximum file size that can be created. This gives some protection against runaway programs. Other parameters in ***/etc/security/limits*** and ***/etc/security/environ*** are discussed in 3.4.3, "Disabling the root Userid" on page 26.

AIX has a substantial number of world-writable directories. Most of these have the *sticky bit* set. In these cases, the *sticky bit* provides the only effective security. Do not remove it!

With AIX, only *root* can use the **chown** command to change the owner of a file. This is more restrictive than some older systems, and may cause a few complaints. The change was absolutely necessary for effective security.

AIX allows file names up to 255 characters long. Not all UNIX systems permit long file names, and excessive use of long names can create problems when interchanging files with other systems.

Many programs create and use work files in the ***/tmp*** directory. This is not very secure since this directory is not protected, and anyone can read these files. The line *TMP=$HOME/tmp* in a user's ***$HOME/.profile*** may help. Some packages and commands use the *TMP* environmental variable to place temporary files. (The user should create the subdirectory ***tmp*** in his home directory, of course.)

Note there is an AIX command named **test**, so users should avoid creating files named "test".

We repeat that AIX does not support *suid* for shell scripts. That is, the *suid* bit in the permissions for a shell script file is ignored. A shell script cannot be run as *root* unless it is executed by a user running as *root*. This is a change from previous systems and is a general security improvement.

---

[49] This is not completely true. A user must have permission to traverse the symbolic link, and this is provided by search/execute permission in the symbolic link.

4.6.1 Unowned Files

Unowned files typically occur when users are removed from the system. When a user is removed (through **smit**, for example), all his files (and his home directory) remain. These files are listed by the system (with the **ls** or **li** commands) with a numeric UID rather than a user name.[50] The user may also have files in other areas; for example, spool files and mailbox files. The **find** command can be used to list all files owned by a specific user (before they are deleted of course).

Using the command **find / -user username -print** produces a list of all files owned by *username*. These files can then be checked, and useful ones allocated to other users (using **chown**). The remainder can then be deleted.

To check a system for unowned files, use **find / -nouser -print**. The listed files can be checked and reallocated (**chown**) or deleted as required. Be careful - some system files will be included in this list, notably ***/dev/console***. DO NOT delete these!!

If NFS is used for remote file system mounting but NIS (Network Information Services) is not used, difficulties can arise in identifying file owners. This is because files that belong to a user on system A, when mounted on system B and viewed by a user on that system, appear to be unowned unless the user on system A is also known to system B.[51] Again, a file is considered unowned if the system (using an **ls** command, for example) displays a number instead of a user name for the owner.

To prevent the **find** command from searching file systems mounted through NFS, the option **-fstype jfs** can be added. Thus, the **find** commands above would become respectively, **find / -user username -fstype jfs -print** and **find / -nouser -fstype jfs -print**.

4.6.2 The /tmp Directory

Many applications place files in the ***/tmp*** directory. Some applications fail to delete these files when they end. Part of the system administrator′s routine should be to check ***/tmp*** regularly and delete those files that have not been accessed within the past few days. Fortunately, **skulker** performs this task on a daily basis and (unless it is disabled) the care of the ***/tmp*** directory can be largely left to it.

The normal mode for ***/tmp*** is that all users can read all files in the directory and all users can create and write new files, but only the owner of a file can delete it. This is done by setting ″other″ (or ″world″) access to *read* and *write* and turning on the ″sticky bit″.[52] However, some applications may require that the ″sticky bit″ for the directory be removed, thus allowing complete read/write/delete access to the directory for all users. This should be avoided if possible, but if it cannot, then all users should be made aware of the potential for damage if files are stored in the ***/tmp*** directory.

---

50 If you add a new user with the same UID as the deleted user, the new user immediately becomes the owner of the files. This UID is the number in the ***/etc/passwd*** entry. A number will not be reused by **smit** unless you force it.

51 The owner may be incorrectly identified in some cases if conflicting user numbers exist on separate systems.

52 This use of the sticky bit is not standard across all AIX-type systems. The original use of the sticky bit is with an executable file; it tells the system to keep the program in storage after it ends.

Confidential, sensitive, or essential data should never be kept in the ***/tmp*** directory as this directory is readable by all users. This is not easy to manage since many program packages automatically place work files in ***/tmp***.

If space is required in ***/tmp***, then (provided your applications do not have essential files there, which they should not) it is safe to delete any files in the directory to create space. That is, it is safe from an operating system point of view; you may have unhappy users, but they should not leave useful data in ***/tmp***.

Chapter 5. Network Security

Connecting a computer to a network, whether it is a Local Area Network (LAN) or Wide Area Network (WAN) opens new categories for consideration when working with system security. For practical purposes, network security can be divided into these areas:

- TCP/IP connections. There are two subcategories for this:
  - Totally in-house LANs, in which there is no possibility of a connection to the larger world of TCP/IP (such as an Internet connection).
  - LANs which have, in one way or another, a connection to “external” systems.
- Dial-in ports for ASCII terminals.
- uucp network operations. (In a sense, this is a subset of dial-in connections, but for practical purposes uucp should be considered separately.)
- All other connections, including SNA.

5.1 Physical Communication Security

Network security involves both physical security and logical security. Other than the following comments, physical network security will not be discussed in this document. We must emphasize, however, the fundamental exposures of physical access to a communications channel. The exposures include:

- Ethernet promiscuous mode. In general, many ethernet (and IEEE 802.3) adapters provide a method of monitoring all traffic on their LAN. Many TCP/IP packages provide a separate module to conveniently use this function. Monitoring can be from any location on the LAN. Anyone with a small PC, the appropriate software, and a connection to the LAN can monitor all data traffic on the LAN. The LAN connection might be an established one (at a connector on an office wall, for example) or an actual “tap” in the LAN cable.
- Token ring performance and monitoring adapter. This is a standard IBM product that can display all the traffic on a LAN. It can be used from any location on the LAN. (In principle, this adapter can be purchased by anyone, but, in practice, it is not widely available.)
- LAN analyzers. These can display all traffic on a LAN. They can be used from any location on the LAN.
- “Data scopes” for SNA/SDLC/HDLC lines. In general, these require RS-232 (or similar) interfaces and are used only at modem locations. It is possible to “tap” a line with a receive-only modem arrangement and use a data scope for monitoring the line.
- ASCII, start-stop monitoring. Many common PC modems can operate in a receive-only monitoring mode. This can be used to “tap” a dial-in session anywhere the baseband telephone signal can be found, such in local telephone wires, wiring closets, and so forth.

Newer LAN topologies, using various types of switched hubs and “virtual LANs,” have the potential for much better security using modes in which session traffic is seen only by the sending and receiving nodes. This technology is quite new

© Copyright IBM Corp. 1994

and will not be in common use for several years. It is being driven by performance factors; the improved security is a side effect.

Wireless LANs, so far, tend to use transmission techniques that are not readily monitored by anything other than another wireless LAN adapter from the same vendor. Given this, wireless LANs have the potential for security problems, although no major problems have been pubicly reported.

While most current LANs are exposed for data monitoring, monitoring on a busy LAN is not a trivial job. In practice, local programming is needed to filter and extract useful data. Such programming is not difficult, but it does require a certain amount of skill and effort and time.

Exposures

The net effect is that all unencrypted data on almost any network can be read ("monitored") by anyone willing to invest the effort and money required. Some monitoring, such as ethernet on your local LAN, requires very little effort or money. Encryption is the only general method of protecting confidential information on a communications network, and general-purpose encryption solutions are not readily available for most day-to-day situations.

5.2 Network Security Goals

Any approach to network security must be built around reasonable goals. "Our network systems must be completely secure" is a nice goal, but not a reasonable one. Practical network security involves managed risks, and should be approached from this point of view. There are two very distinct security areas:

1. Confidentiality of session data, to and from your system, that appears on the network.
2. Authentication and access control (for login, file transfer, remote commands) for your system.

As mentioned earlier, strong confidentiality of session data is difficult (or impossible) to obtain in network environments. Within a limited and controlled environment, this risk can usually be tolerated. In larger environments (with lesser-known users) the risk grows and may not be acceptable. This means that network topology and segmenting becomes an important security factor. Smaller LANs, with some degree of isolation between LANs, lower the risks associated with monitoring. This leads to a demand for *firewalls*, which are systems installed to limit the types of traffic between two LANs.

New technology may resolve some of the worst exposures of network security. DCE, for example, has totally encrypted login sequences and an option for encrypted session traffic. With today's DCE, these functions can be used only for interactions with DCE servers. Within this context, DCE can provide secure and confidential communication throughout a network. Another new technology, mentioned previously, provides switching in a LAN hub such that LAN nodes do not see (and cannot monitor) packets that are not addressed to them.

5.3 The securetcpip Command

Some TCP/IP commands provide a relatively secure authentication environment during their operation. These commands are **ftp**, **rexec**, and **telnet**.

These commands provide security functions only for their own operation. They do not provide a secure environment for other commands. For example, a user can **telnet** to another system (with reasonable authentication security provided by **telnet**) and, once logged into the remote system, do something that is completely insecure.

The **securetcpip** command disables less-secure "standard" TCP/IP commands. We recommend that you use the **securetcpip** command on all systems in your network unless there is a strong requirement for the commands it disables. **securetcpip** is a shell script that disables commands and daemons by editing out the relevant stanzas in ***/etc/inetd.conf*** and using **chmod** to set the permissions for the executable commands to 000 (---------). No user can run the commands once this occurs.

Before running **securetcpip**, you should quiesce any networking programs that are running. If the various daemons have been started using SRC (the System Resource Controller), they can be stopped by:

```
stopsrc -g tcpip
```

This command will stop all TCP/IP related daemons. You can then enter:

```
securetcpip
```

After running **securetcpip**, the following commands and daemons will be disabled and become unavailable for use:

- Daemons
  - **rshd**
  - **rlogind**
  - **tftpd**
- Commands
  - **rlogin**
  - **rcp**
  - **rsh**
  - **tftp**
  - **trpt**

securetcpip disables the use of these commands. It is a reversible change, and the commands can be re-enabled if required, by uncommenting the stanzas in ***/etc/inetd.conf*** and changing the permissions on the programs and daemons so that they can once again be executed. To prevent this, you might consider deleting the relevant commands and daemons. (In practice, few installations do this.)

The **securetcpip** command creates ***/etc/security/config***, which contains stanzas that restrict ***$HOME/.netrc*** usage by **ftp** and **rexec**.

After doing this, your users must use **telnet** instead of **rlogin** or **rsh**, **ftp** instead of **tftp** and **rcp**, and **rexec** instead of **rsh**. This provides better control of network security and is a basic step to prevent unauthorized users from accessing your system.

Note: XStations may use **tftp** to download the X Windows server code from AIX. You will need to verify that your XStations can operate without **tftp** before executing **securetcpip**. This restriction may also apply if you have diskless workstations in your network.

5.3.1 Remote Login Controls

Using TCP/IP connections and commands, UNIX users can log into remote systems. A variety of commands, already mentioned, can be used. If no special steps are taken, these remote logins follow normal security controls; that is, the remote user must supply a valid local userid and password.[53] It is possible to trade security (local userid and password required) for convenience (skip the login process when connecting to another system). The ***/etc/hosts.equiv, $HOME/.rhosts***, and ***$HOME/.netrc*** files are used to control this trade-off.

Should you permit the use of these functions (described below)? This is a difficult question. The convenience factor can be important. If your users frequently connect to each others′ systems (using **rlogin**, or something similar), skipping the login sequence each time is very convenient. However, the potential security exposure in allowing this is large.

We recommend that you permit these functions only if your network is small (all users known and trusted) and not connected to larger networks. If your network is connected to much larger networks, we recommend that you run **securetcpip** (providing the most protection), or at least prohibit use of the files discussed here (less protection). If you do not use **securetcpip**, but do prohibit the following files, remote users can use **rlogin** (and associated commands), but must provide a userid/password for each system. If you use **securetcpip**, remote users can still connect to your system, but they must use **telnet** (or **ftp**) and supply a userid/password.

Please note that this complete discussion does not apply to X Windows connections. Security considerations for X Windows connections is briefly discussed in 9.2, “X Windows” on page 92.

The /etc/hosts.equiv File

The ***/etc/hosts.equiv*** file is used by the **rcp**, **rsh** and **rlogin** commands to bypass local authentication. Only one such file, with this specific name, may exist in a system. If a remote host tries to execute one of these commands on the local system, the local system will check the ***/etc/hosts.equiv*** file to see whether that host is listed in this file.

The file takes the form of a simple list of system names. A typical example might be:

```
hosta
hostb
hostc
```

If the remote host is in this list, no local authentication (login, password) is required by the remote user if a local account with the same userid exists. The remote user is automatically logged into the local system as the local user with the same userid. For example, if “hostb” is listed in the local ***/etc/hosts.equiv*** file, and if user *joe* has an account on the local system, then a user *joe* on

[53] This userid and password flow, in the clear, over your LAN connections. They have the normal LAN monitoring exposures.

system “hostb” can connect and automatically log into the local system as *joe* without supplying a local password.

The ***/etc/host.equiv*** is not checked if the user is *root*. That is, *root* must provide his password even if an ***/etc/host.equiv*** entry exists for the connecting system. If an ***/etc/host.equiv*** entry exists for a connecting system, no check is made for ***$HOME/.rhosts*** files.

The ***/etc/hosts.lpd*** file provides a similar service but is limited to those hosts requesting a remote printing service. If a remote system is defined in ***/etc/hosts.equiv***, it can issue commands and send printing to the local system. If the system is defined only in ***/etc/hosts.lpd***, it can only submit print jobs and is unable to run commands.

You (the administrator) can control these two files because they must be in the ***/etc*** directory, which should be protected.

The .rhosts Files

A ***$HOME.rhosts*** file provides a function similar to that of ***/etc/hosts.equiv***, but more limited in scope. Multiple ***.rhosts*** files can exist, in various $HOME directories. They permit a remote user to log into a local account without local authentication. The format is:

```
remote_hostname   userid
```

For example, if user *fred* has an ***.rhosts*** file in his home directory. ***/u/fred/.rhosts*** might contain the following lines:

```
sysa steve
sysa juliet
sysb clare
```

The remote users *steve* and *juliet* on host sysa and *clare* on host sysb can log into the local system as *fred*. AIX will not use ***.rhosts*** files that have permissions permitting anyone other than the owner to write to the file; such files are simply ignored. (This limitation would be even better if it also ignored files which others could read.)

If these files exists on your system, a user defined in one of these files can login or execute commands without having to provide any form of user authentication. They can execute commands or login with the same permissions as the local user whose home directory contains the ***.rhosts*** file.

You cannot easily control these files because any user can place them in his home directory. You can, effectively, disable the use of these files by disabling **rlogin** (and his friends) with the **securetcpip** command.

The .netrc Files

These files (in various users′ home directories) allow processing of **rexec** and **ftp** commands without manual password verification. One of these files can appear in any user′s home directory. A typical entry might be:

```
machine tardis login mike password cyberman
```

This will present the userid *mike* and the password ″cyberman″ when the local user attempts to **ftp** or **rexec** to the remote host called ″tardis″, thus removing the need for manual authentication on the remote system.

Note that these files control outgoing connections, whereas ***/etc/hosts.equiv*** and ***$HOME/.rhosts*** control incoming connections.

A .netrc file contains unencrypted password information, and is a serious security exposure. You should avoid having .netrc files on your system because these userids and passwords expose other networked machines. If you must have these files, it is important that the file has its access permissions set so that ONLY the owner can read and write to it. (600 or rw-------). Some TCP/IP functions attempt to enforce this; if the permissions on the file are not set to 600, an automatic login will fail.

If the **securetcpip** command was used on your system, it created ***/etc/security/config***. This file contains stanzas limiting the use of .netrc files.

5.3.2 Other Important TCP/IP Files

These files control system-level TCP/IP functions and are critical elements of security.

The /etc/hosts File

This file describes the network hosts that the local system identifies by name. The file matches a name to an IP address. A typical example might be:

```
9.12.2.32     gateway
9.12.2.95     bill
128.100.1.4   dtp
```

The ***/etc/hosts*** file is used only if a *name server* is not active, or if the name server is unable to resolve a name. A name server, or the ***/etc/hosts*** file, permits users to refer to systems by name instead of by their dotted-decimal IP address. Even if a name server is normally used, the ***/etc/hosts*** file should be secure because it will be used if the name server fails. Only the administrator should have *write* access to this file. In general, *read* access is permitted for everyone.

You should check ***/etc/hosts*** at regular intervals to ensure that names are listed with the correct IP addresses.

The /etc/inetd.conf File

This file enables and disables TCP/IP services. The **inetd** daemon starts other TCP/IP daemons when particular services are required. For example, if a user uses the **telnet** command, the **inetd** daemon starts the **telnetd** daemon to handle this request. Services can be withdrawn from the TCP/IP environment by removing the relevant stanza in this file. This file provides the primary control point for managing what TCP/IP services are available on a system.

Name Server

If your system is a name server, you must protect the name server data files. File ***/etc/resolv.conf*** should be protected on all systems. Incorrect data could direct a system to an unauthorized name server. Files ***/etc/named.boot, /etc/named.ca, /etc/named.local,*** and ***/etc/named.data*** should be protected. (Other file names can be used by a name server, but these are the normal names.)

5.3.3 The netstat Command

netstat provides network status information. It is commonly used when diagnosing network problems. It can also provide information that is useful to check network security. For example:

```
netstat -p tcp
```

provides information about the TCP/IP protocol since booting the system. Look for things such as failed connection attempts. This may mean someone is trying to break into your host. There are many options for **netstat**, and several may be useful for security purposes.

5.4 Network File System Overview

The Network File System, or NFS, provides a method of accessing files and directories on other machines on the network and treating them as if they were local. A directory on a remote machine can be **mounted** onto the local filesystem. It then appears to be local to the users. All actions performed on this mounted directory are automatically passed across the network to the remote host for processing.

Because NFS uses TCP/IP protocols and not TCP/IP commands to provide its facilities, most of the precautions we have mentioned so far will have little effect on an NFS network. For this reason, we must regard NFS security as a separate topic.

There are two basic components in an NFS network, the *server* and the *clients*. The server exports directories to the clients. The directories that may be exported and the permissions or restrictions associated with them are listed in the server′s ***/etc/exports*** file.

The client system must **mount** the exported directory into its local file system before it can use it. On a client system, issuing the **mount** command will show the mounted filesystems. Remote file systems will have an entry in the ***node*** column. On a server system, you can see which hosts have remotely mounted directories by using the **showmount** command.

It is important to understand the use of the term "server" in the context of NFS. Potentially, any system can be an NFS server. For example, a group of workstations (with a common interest in a project) might all "export" certain directories to each other. That is, every workstation in the group might "mount" selected directories on other workstations in the group. In this case, all systems are both NFS clients and NFS servers.

Stated another way, you may have formally identified NFS servers that probably have large disks and are used by everyone in the organization. You may also have informal NFS servers active (perhaps as described in the last paragraph) that are intended for use within a smaller group. You (the administrator) may not be aware of some of these more informal NFS activities.

Mounting a remote directory is not necessarily automatic, although placing the relevant stanzas in ***/etc/filesystems*** can solve this. Automatic mounting should be avoided unless user access is carefully controlled. It is much better to use a shell script or commands to mount remote directories than to automate them into the startup routine. This can prevent directories being mounted unnecessarily and so somewhat reduce security exposures.

NFS-exported directories will not protect information if the administrators of the NFS client systems are not trustworthy. Someone with the ability to create users and groups on an NFS client system can "see" any file in the NFS-exported directory. Stated another way, for an NFS environment to have any security whatsoever, the administrators on all server AND client systems must be trustworthy. Since workstation owners are usually their own administrators, this implies that all workstation owners involved must be trustworthy (in the sense of trustworthy security administrators).

5.4.1 The /etc/exports File

The list of directories that can be exported from a server is maintained in the ***/etc/exports*** file. Each directory must have a separate entry. You (the system administrator) directly edit this file. A typical entry might be:

```
/usr/custdata -access=clienta,ro
```

Which allows the directory ***/usr/custdata*** to be exported only to the system "clienta". It is exported read-only so that changes cannot be made by the remote system.

It is important to restrict access to any exported directory. **The default, with no options specified, is to export a directory to all users with read-write permissions!** A variety of controls are available within the ***/etc/exports*** file, including control of *root* access via NFS.

As an administrator, you have two particular concerns with ***/etc/exports*** files:

1. For your "official" NFS servers, do the ***/etc/exports*** files provide sufficient protection? Are only needed directories exported? Could lower-level directories be used instead?
2. For other hosts, do ***/etc/exports*** files exist? If so, they imply that that system may be functioning as an NFS server. Does this comply with your security policies?

The directories listed in ***/etc/exports*** are not available until the **exportfs** command is run. This command is run by the ***/etc/rc.nfs*** shell script at startup time and should be run again for any change to ***/etc/exports*** to become active. When **exportfs** is run, it interrogates ***/etc/exports*** and updates the file ***/etc/xtab***. This maintains the list of currently exported directories. Do not edit ***/etc/xtab*** directly; use **exportfs -a** to update ***/etc/xtab*** .

Do not export directories that do not need to be exported. Always try to export the maximum path (that is, the most detailed directory specification) required for the task.

Always ensure that the permissions that are set for exported directories are as limited as possible. For example, if you wish to export ***/usr/lpp/appl1/bin*** and ***/usr/lpp/appl2/bin***, you should have two entries in ***/etc/exports***:

```
/usr/lpp/appl1/bin
/usr/lpp/appl2/bin
```

You should not have the higher level directory entry:

```
/usr/lpp
```

5.4.2 NFS Support for ACLs (Access Control Lists)

NFS under AIX supports the AIX filesystem ACL model. There is an RPC layer program that passes ACL information between the client and server. This may lead to some unexpected results.

ACL support is an additional function that has been added to the AIX Version NFS and, as such, it does not change the NFS protocol specification. Other NFS clients may not support ACLs and cannot see them. This may lead to problems when the ACL permissions and permission bits differ if the client is a non-AIX system. Such problems should not arise if the client is also an AIX system.

Do not use ACL functions in heterogeneous networks.

5.4.3 Secure NFS Operations

NFS defines a method for "secure" operation. Properly used, this function provides secure authentication of NFS users. It does not provide additional data protection, encryption of LAN traffic, or encryption of data. It does provide protection against counterfeit host connections (that is, a system that claims another systems name and/or IP address). In practice, the Secure NFS function is not often used.

To use Secure NFS, you must install and use NIS. You must change a few of your normal operational procedures, such as using **keylogin** instead of **login** and **yppasswd** instead of **passwd**. You must maintain several files (including ***/etc/hosts***, ***/etc/keystore***, and ***/etc/.rootkey*** in every system) in consistent states. You should install time daemons in every system. **Do not use Secure NFS without studying it well. Build and use a trial environment before using it in a larger production environment.** The following is a brief overview of the Secure NFS function. Consult the full AIX documentation for more detail. See Figure 2 on page 71 and Figure 3 on page 72 for an overview of part of the Secure NFS process.

To mount a directory for Secure NFS, both ends of the connection must be set up correctly. On the server, the ***/etc/exports*** file must have an entry for the directory to be exported with the **secure** option set thus:

```
/usr/secretdata -secure
```

where ***/usr/secretdata*** is the exported directory. Other restrictions on the directory may also be specified.

The directory should be exported using the **exportfs -a** command. It is worth checking that the ***/etc/xtab*** file also shows that the directory is to be exported with the **-secure** option. If the option is set to **-access=secure**, the mount will not work.

On the client there must be an entry in ***/etc/filesystems*** for the securely mounted directory. This entry is of the form:

```
/usr/secretdata:
       dev        = /usr/secretdata
       nodename   = rs6000
       vfs        = jfs
       mount      = true
       check      = true
       options    = secure
```

Other options, such as **ro**, can be specified as well. If the ***secure*** option is not specified, DES authentication will not be used, and the directory will be exported using the standard AIX authentication scheme.

If you wish to use this **secure** mount, you must configure DES authentication on your system first.

DES authentication security is based upon the ability to encrypt the current time which the receiver then compares with the encrypted system time on its machine. If **timed**[54] is running and there is a synchronized network time, this is performed automatically. If not, the client asks the server for its time and then calculates by how much the client system time differs and does an appropriate time conversion.

Secure NFS also uses a public key system, with public and secret (private) keys. The public key system is used to establish initial DES keys for a session. The public keys are established with the **chkey** command or by the system administrator with the **newkey** command. This prompts for the user′s password and then generates an encrypted key-pair containing a public key and secret key.

The client and server must share the same encryption key to begin a conversation. The client generates a random *conversation key*, used to encrypt the timestamp. This is calculated from public and secret keys in such a way that the the server does not need to know the client′s secret key and vice versa. Only the client and server can calculate this because doing so requires knowing one secret key or the other. An outside agent cannot calculate the secret key by any reasonable means.

The conversation key is encrypted using a public key and sent within a packet known as a ***credential***. This is the only time a public key is used for encryption.

The credential contains the name of the client, the encrypted conversation key, and a variable window that is itself encrypted by the conversation key. The window contains the encrypted timestamp and an encrypted verifier for the window. This makes it much harder to guess the credential. The server stores this information into a credential table and replies to the client. In the returned verifier, the server sends an index ID to the credential table plus the client timestamp minus one, again encrypted by the conversation key. Because only the client knows the original timestamp that was sent within the credential, only the client will know the value of the timestamp minus one. Hence, the client can verify the server. This occurs whenever service is requested by the client system.

Note that the encryption key is sent back and forth only at the start of the conversation. It is not sent on each client transmission. This makes it much harder to determine the key and "break the code."

This whole process is only to verify the identity of the client and server. User data is not encrypted.

[54] This is a TCP/IP related daemon that synchronizes the system time across multiple systems.

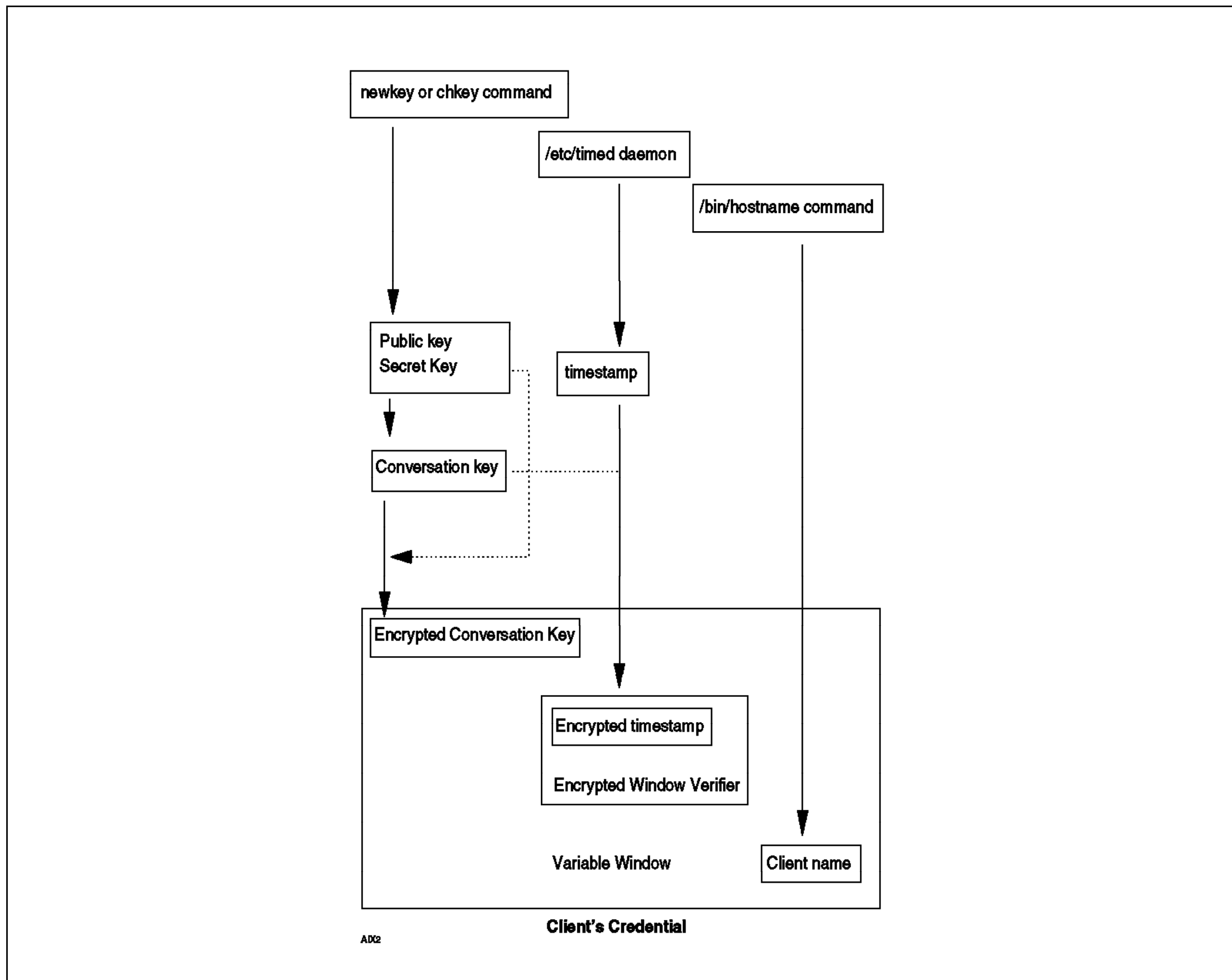

Figure 2. NFS Client Credential

5.4.4 The Client - Server DES Interaction

The Secure NFS service relies on the DES encryption facility which in turn requires public and private keys. These keys are usually generated for the user from that user's login password. To remain synchronized, the password is updated in a central point in the network maintained by NIS and the ***yppasswd*** command.

5.5 Network Information Service (NIS)

The creation of a single systems image, where every machine has a userid for every user and the work environment for users is the same regardless of which machine in the network they are physically connected to, is possible using NIS. NIS was known as *yellow pages* or *yp*, but these terms are not used in current systems.

NIS shares *system* information. Without it, having a large number of systems on a network would create problems for the user. NFS can be used to make a user's files and directories appear on every system. However, since NFS does not translate user or group IDs between systems, every system would need an ID for every user. This problem can be solved using NIS.

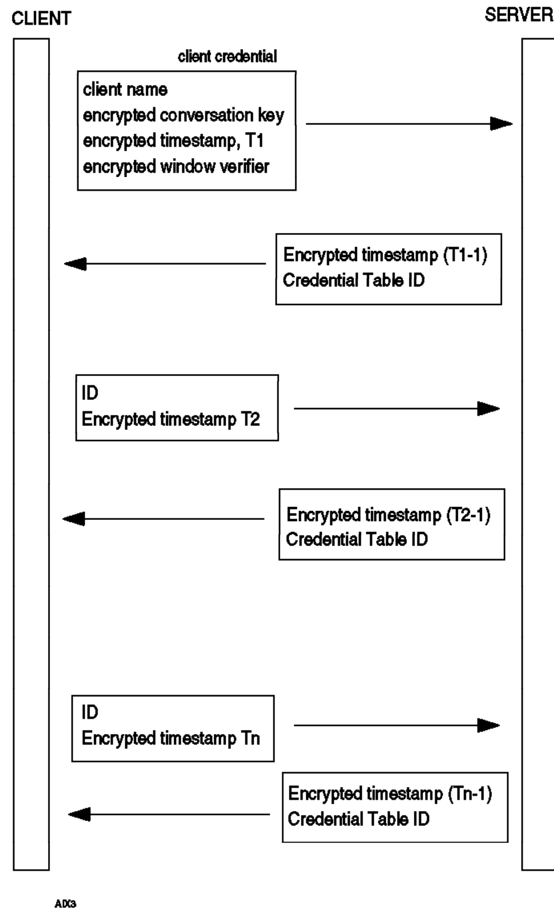

Figure 3. NFS Client-Server DES Interaction

Using NIS, one system is chosen as the network server for files such as ***/etc/passwd*** and ***/etc/group***. This then becomes the NIS server and updates are made only to these server files. Client systems request this information from the server.

Using NIS to administer a network can create security exposures, because the administration files kept on the server are readable from any networked machine. For this reason, the selection of NIS as a network management tool should be made with the utmost care and a full understanding of the implications in such a step. The centralized files also provide intruders a centralized point for attack.

It is possible to establish and maintain a secure NIS environment, just as it is possible to establish and maintain a secure UNIX environment. Both require continuing administrative effort and considerable attention to detail. Security risks with NIS are perhaps greater because a single failure can expose every system in the network.

5.6 Adapter Security Levels

An earlier version of AIX provided specialized security controls at the LAN adapter level. This permitted, for example, an administrator to specify that only *top secret, research* data might be sent on a specific LAN connection. Part of these functions were described as *data import security* and *data export security*.

These functions are not present in AIX 4.1. They were only partly implemented in the earlier AIX releases, and were in anticipation of TCP/IP changes that did not occur. The TCP/IP community has selected other methods for future security improvements (as part of the planned migration to 16-byte address fields).

Chapter 6. Logs and Accounting

AIX has the "standard" UNIX accounting subsystem. This is not enabled when AIX is installed. AIX accounting can be used for several purposes, including:

- Charging users for services
- Monitoring system activity and resource utilization
- Investigating security incidents

There are costs involved in using the accounting system:

- It adds overhead to the system. This can be nontrivial if process and disk-usage accounting are active.
- Someone (we assume this is you, the administrator) must take time to understand the accounting system, including its files, cron processes, and controls. Over time, small problems occur with the accounting files and someone must monitor and correct these problems.
- Someone should use the output; otherwise there is little point in collecting it.

Charging users for computing resources used was the primary purpose of the accounting process, and an important activity in the earlier days of UNIX. Todays trends -- to less expensive, distributed processors -- have generally negated the need to charge users. The use of UNIX accounting for charging users (directly or indirectly) is quite rare for workstations, and is often not used even for larger multiuser servers.

From a security viewpoint, our recommendations are:

- Do not activate the accounting processes solely for security.
- Do become familiar with the usage log files that AIX writes, even when accounting is not active. (These are described below.)
- For a larger multiuser server, learn how to activate all the accounting functions (and how to deactivate them). If you suspect intruders in your system, you can enable accounting for a closer look at system activity. (For a still closer look, you can use the auditing functions described in a later chapter.)

The AIX *System Management Guide* contains a full section dealing with starting and using the accounting system. Read this thoroughly before activating the accounting system.

6.1 AIX Log Files

AIX automatically writes several log files, regardless of whether the accounting system is enabled. These can be very useful for day-to-day security monitoring. In most cases, simple commands are provided for displaying the files. The key files and associated commands are:

- ***/var/adm/wtmp*** contains an entry for every time a user logs into or out of the system, plus entries for some system processes. It also contains a record of system restarts. The accounting process (if active) uses this file, and will remove data after it has been processed by the accounting programs. If accounting is not active, this file grows until you shorten it. A simple and useful command for viewing this data is **last**. A number of examples are

given below. Do not delete this file. If you want to restart it, copy ***/dev/null*** to it.

- ***/etc/utmp*** contains entries about currently active users and subsystems. When a user leaves the system, he is no longer reflected in this file. The basic command used with this file is **who**. Over time, this file may grow due to accumulated errors, and you may need to restart it. Do not delete it; copy ***/dev/null*** to it instead.
- ***/var/adm/sulog*** contains a simple ASCII record for every time the **su** command is used. It can be listed with the **pg** or **cat** commands, or with any text editor. In time, you will need to restart or truncate this file.
- ***/etc/security/failedlogin*** contains entries for login failures. It can be displayed with the **who** command.
- ***/etc/security/lastlog*** contains a stanza for every system user. Each user′s stanza contains information (time, port, host) for the last successful and unsuccessful login for this user. This is an ASCII text file. Unfortunately, the time stamps it contains are very large numbers containing the number of seconds since January 1, 1970, which are not meaningful to a human reader. There is no standard program to redisplay the information in a more human format, but it would not be difficult to write such a program.

The logs described above may not be effective against a user running as *root*. These log files will contain the same entries for *root* as for any other user, but *root* can manipulate the files and hide his activities. Hiding all traces of *root* activities is not especially simple, but a skilled user can do it. Conversely, a lesser-skilled person would not know how to do this, and log entries will exist for his activities.

The ***/var/adm/wtmp, /etc/utmp,*** and ***/etc/security/failedlogin*** files all have the same format. It is described in ***/usr/include/utmp.h***. You can write relatively simple local programs to produce customized reports from these files. The **who** and **last** programs are provided to display these files, with many selection options. The **last** command has fewer options, but is useful because it lists output in reverse order, with most recent entries first. (This is quite useful if you are displaying a large ***/var/adm/wtmp*** file.)

By default, **last** displays data from ***/var/adm/wtmp***, and **who** displays data from ***/etc/utmp***, but either command can be used with the three files in *utmp* format.

Typical commands might be:

```
who -a /etc/utmp                      (current users)
who -a /var/adm/wtmp                  (complete history)
who -a /etc/security/failedlogin      (complete history)
```

These commands format and display everything in the files. This includes many lines about system-started processes that may not interest you. The forms:

```
who -u /var/adm/wtmp
who /var/adm/wtmp
```

generate less output and may be more useful for browsing the data.

Chapter 7. Trusted Computing Base

> *The first release of AIX 4.1 may not have the trusted computing base available. The information in this chapter will apply to later releases.*

The Trusted Computing Base (TCB) of AIX can be confusing. It is all of the following:

1. A concept
2. Some programs, such as **tcbck**
3. Control files
4. A flag in selected files
5. A trusted login process
6. A trusted shell
7. An installation option

You are not *required* to do anything with the TCB. AIX runs quite well if you ignore it. However the TCB, in conjunction with the **tcbck** command, provides very useful tools for both security and system integrity. The integrity consideration may be the most important to you; the TCB facilities can help detect or prevent accidental system changes and help protect you from playful, knowledgeable users.

There is an installation option for the TCB. *We recommend that you always install the TCB, on every system, even if you have no immediate plans to use it.* The additional installation time and disk space is trivial, and the TCB cannot be added after AIX 4.1 is installed.

Based on your organization's security policy, you (or your management) must decide how secure your system should be. The TCB facilities can help you maintain a reasonable level of assurance about system integrity. The TCB functions do require a modest effort to understand and use, and provision must be made for this time and effort. You might use the TCB functions only to verify that the base AIX components are correct. With more effort, you can use them to monitor and verify that your key applications are intact and secure. In some cases you might want to use (or make available to many users) the secure login and secure shell facilities. Your degree of use of TCB functions will determine the time and effort required to understand them.

One exposure that the TCB functions cannot protect you against is an authorized (you permitted it to be installed) *suid root* program that (accidentally or by design) gives improper access to users. TCB functions can locate unauthorized *suid root* programs, but they cannot verify the internal logic of any program.[55]

---

[55] The AIX audit facilities can be used to trace critical system calls within a program. Audit is discussed in a later chapter.

© Copyright IBM Corp. 1994

7.1 TCB Description

A trusted computing base is the set of programs and files that must be correct ("trusted") if the rest of the system is to have security and integrity. This includes programs such as the AIX kernel, the **login** program(s), the **passwd** program, and so forth. Likewise, the ***/etc/passwd*** file and other key control files must be correct. In addition, there should be a method for a user to connect to the system and assure himself that he is communicating with the proper **login** program(s) and not a counterfeit. Likewise, a shell should be available that is known to be correct and that operates only in a correct environment.

The AIX trusted computing base provides these functions, with tools to ensure that the TCB remains intact.

Any trusted computing base starts with the *assumption* that the basic system delivered by the vendor can be trusted.[56] That is, the AIX TCB assumes that IBM has delivered a system in which the key system components provide proper security and integrity. You can add any programs or files you choose to the TCB, and, in production server environments, there are excellent reasons for doing this. (Not all the components of AIX are in the TCB; only a relatively small percentage of the programs and files are within the TCB.)

The most useful function of the TCB, from an administrator's viewpoint, is the checking process associated with it. It can maintain, in ***/etc/security/sysck.cfg***, a list of the attributes (permissions, owner, checksum, links, and so forth) of various files. (The database is not restricted to files in the TCB.) The **tcbck** command can then check that the files still have these attributes (and, optionally, correct them if something changed. This is a fast, complete way to verify that all the programs/files that form your TCB (plus any other files you include) still have the proper attributes and are unchanged.

You should examine ***/etc/security/sysck.cfg*** (using the **pg** command) to better understand the attributes it can record.

AIX defines a TCB bit in inodes. This bit indicates that the file is part of the TCB. A trusted shell (briefly described later) is part of the TCB and will execute only programs that are part of the TCB. An inode TCB bit is used to indicate that a file is part of the TCB and can be executed by the trusted shell (or on a trusted path). The TCB bit can be set (by *root*) using the **chtcb** command. The only purpose of the TCB bit is to control which programs can be executed from the trusted shell. There is no point in setting the TCB bit in your applications unless you need to execute them from the trusted shell.

7.2 Using the tcbck Command

If you install AIX with the TCB option (as we recommend), you should find an initial ***/etc/security/sysck.cfg*** file that matches the installed system. You can check this with the **tcbck -n ALL** command. (The command may report minor problems, which you can correct or ignore. For example, device (such as ***/dev/pty***) ownership and modes will change as users log in and log out of the system.) This command reads the ***/etc/security/sysck.cfg*** database and verifies

[56] This assumption may be based on tests provided by other organizations, on the known history of the product, on assurances provided by the vendor, and so forth. In any event, the starting point is the vendor-supplied operating system.

that every file listed in the database has the same characteristics that are listed in the database.

It is not practical to regenerate the complete ***/etc/security/sysck.cfg*** file. You can add your programs and files to it, but the initial creation (covering hundreds of trusted files in AIX) is supplied with AIX. If you plan to use the **tcbck** function, you should maintain (and update, if required) the configuration file that is supplied with AIX.

One option of the **tcbck** command causes it to "walk the directory tree," examining all files in your system.[57] It will list any files which should be considered part of the TCB; these include, in particular, all *suid root* files.

The **tcbck** program has options to check the files listed in its database and automatically correct attributes that do not match the attributes listed in the database. These are the "p" and "y" options. *We strongly recommend that you not use these options.* We recommend that you manually resolve any listed errors, so you understand what you are changing.

Earlier releases of AIX used the **sysck** command to perform the functions now covered by the **tcbck** command. AIX 4.1 contains a **sysck** command, but it now is related only to software installation functions. You (the security administrator) should have no need to use **sysck**.

You should run **tcbck** periodically, simply to verify the integrity of the base system. You should consider adding your production programs (if they are relatively stable) to the data base. If you are very concerned about system integrity, you might copy the database (***/etc/security/sysck.cfg***) to a diskette or tape and restore it before use. (This would prevent anyone from altering the system and then altering the corresponding file attributes in the database.)

Several other "check" commands, described in 3.8, "Verifying the User Environment" on page 33 are closely related to the **tcbck** command.

7.3 Using the Trusted Login and Trusted Shell

The login process, for any UNIX system, can be a major security exposure. Locally attached ASCII terminals and the major graphics terminal of a workstation are especially exposed. The problem is simple: is the "real" **login** program[58] controlling the terminal, or is another program simulating the **login** program? This can be a surprisingly difficult question to answer.

A user, with only modest programming skills, can write a program that clears the screen, presents a login prompt, and waits for input. If this program is left running at a terminal, another unsuspecting user may assume the terminal is free and attempt to log into the system. The running program then captures the userid and password and terminates its session. This causes a new login prompt (from the "real" **login** program). The user may be surprised by this second login prompt, but will usually just login again. (A knowledgable, slightly

---

[57] If you use this option, you might consider unmounting CD-ROM and NFS file systems first. You might want to mount private user file systems so they will be included in the check.

[58] Technically, we should also be concerned about the connection to the **getty** program, but will refer to it as the generic **login program.**

paranoid user will recognize that he has been spoofed and immediately change his password. However, very few users will recognize the situation.)

This is a classic UNIX attack, and can be effective even on modern UNIX systems. AIX provides a defense against such attacks with a secure attention key (SAK) and a trusted path for login. This process is not automatic. The administrator must enable the SAK process for users and for ports. Users must follow a slightly different procedure when logging into the system. Even when enabled, users are not required to use the SAK/trusted path process; they can continue to login as usual (and with the usual exposure to Trojan horse programs).

The SAK has two purposes:

1. If the user is not already logged into the system, it will secure the login path (terminating any login Trojan horses in the process). It will connect the user to the trusted shell (**tsh**) if his *tpath* is defined for it, otherwise it will connect him to his normal shell.
2. If the user is already logged into the system, it will terminate all programs that are connected to his port and connect him to the trusted shell, with a trusted path, if his *tpath* is defined as *on*; otherwise it is ignored.

When a user enters a trusted path, the permissions for his port are changed (automatically, by the SAK-driven process) to 600 (octal) instead of the 622 permissions normally associated with a connected port.

Ports are enabled for SAK by setting the *sak_enable = true* parameter in ***/etc/security/login.cfg***. This can be set in the *default* stanza (making it apply to all ports), or set in stanzas for each port. Once this is set[59], terminals should respond to SAK, which is CNTL-x CNTL-r (hold down the CNTL key and press x, then press r). This change must be done by editing the file; it cannot be done with **smit**. If the port is the main graphics terminal of a workstation, you must add two extra lines to ***/etc/security/login.cfg*** to enable SAK. These lines are:

```
/dev/console:
        synonym = /dev/lft0
```

These lines may already in the file, but are commented out.

Users are enabled for trusted path and trusted shell by setting the *tpath* parameter in the appropriate stanza in ***/etc/security/user***. This can be done, for individual users, using **smit**. It can be set in the *default* stanza by editing the file. The parameter can be set to one of the following values:

1. *tpath=nosak*. This is the default value and means that the trusted path is not available for this user. An SAK is ignored if this user is already logged into the system. An SAK before login will cause the user's normal shell to be started.
2. *tpath=on*. This enables optional use of the SAK and trusted path for this user, and is the value appropriate for the functions discussed in this section. A SAK after login will cause the trusted shell to be started. A SAK before login will cause login to start the trusted shell.
3. *tpath=always*. This permits the user to log into the system only through a trusted path (produced with SAK) and restricts the user to the trusted shell

[59] We found it necessary to reboot after setting the parameter.

ONLY. We recommend not using this value unless you carefully evaluate its restrictions first.

4. *tpath=notsh*. This value will force an immediate logout if the SAK is entered while the user is logged into the system.

The trusted shell, **tsh**, will execute only programs that have the TCB bit associated with their permissions. It will not permit changes to the current PATH, and has a very restricted set of shell commands and variables. One command is **shell**, which terminates the trusted shell and connects the user to his normal login shell. (The **shell** internal command is not effective for *tpath=always* users.)

The SAK and trusted shell, together, provide a useful function for an administrator in an inherently insecure environment (such as a university), for an unusually critical installation or application[60], or for a truly paranoid user. The trusted shell is not useful for "normal" users because its functions are so limited.

The SAK, by itself, is quite useful for avoiding login Trojan horses. You could use it like this: Inform the users of the availability of the trusted path login. A user then has the option to use a trusted path login if he suspects a possible security problem. The process involved is simple. For example, to login under a trusted path (protecting the password) and then drop back to a normal environment, follow the procedure below:

1. At the login prompt, press CNTL-x CNTL-r (the SAK sequence). A new login prompt should appear; that is, the screen should "roll up" a new prompt. This is a signal that the SAK was effective. If this "roll up" does not occur, the SAK was not effective, and a secure path is not established.[61]
2. Log in as usual.
3. If your normal shell prompt appears, continue as usual.
4. If the **tsh** shell prompt appears, enter the command **shell**. This will drop the trusted shell and reinitialize the user's session with his normal shell.

If there is any chance that you, or your users, might use the SAK and/or trusted shell, we recommend that both be enabled. The overhead is slight and, in certain instances, they can provide an important element of security.

---

[60] This is a very subjective evaluation, of course.

[61] The roll up is just your standard herald being redisplayed.

Chapter 8. Auditing Functions

The auditing subsystem provides a means to trace and record security-relevant information. You can use this information to detect potential and actual violations of security policy. You (the administrator, operating as *root*) can configure and control the audit subsystem. By default, the audit subsystem is not enabled when you install AIX.

A number of commands, control files, and parameters interact to control auditing. These can become complex. The following descriptions concentrate on basic usage and assume you are using the audit control file distributed with AIX, with only minor changes.

When you start the audit subsystem, it audits (that is, generates an output record) for *events* and *objects*. Two different recording modes can be specified, *BIN* and *STREAM*. Brief definitions of these terms are provided here.

Audit Events

An *event* is the execution of a specified system module, such as a module that creates a directory or updates a password. Event detection is distributed throughout AIX (within trusted modules). A list of all defined AIX audit events is given in Appendix C, “Audit Events” on page 115. (You can add audit events in your own programs, and extend this list, but this would be unusual.) A module that processes an auditable event reports it to the *audit logger*. The report includes the name of the event, the success of the event, and event-specific data. Through various audit controls you can select which of these events you want to activate and record. You should be as selective as possible when doing this. Recording all possible events for every user in the system can produce a huge amount of data and considerably impact system performance.

Event auditing is ALWAYS associated with a userid (or userids). For example, you can audit every time user *joe* creates a new directory. If the audit subsystem is active AND if new-directory events are being audited for *joe*, an audit event (record) is created. You can list many userids to be audited, with many types of audit events for each user, but you cannot readily “audit everything.”[62]

There are approximately 130 different audit events built into AIX. These tend to fall into related groups. For this reason, audit events are grouped into *classes*. You can define which events are in a class. The class names are arbitrary. It is class names (rather than individual audit event names) that are associated with userids when the audit subsystem is active.

[62] There is an undocumented *default = ALL* line that can be added to the *users* stanza of ***/etc/security/audit/config*** and that will record all classes for all users who do not have specific event class definitions. Events not included in the defined classes will not be recorded.

© Copyright IBM Corp. 1994

Audit Objects

In addition to events, you can audit *objects*. In practice, this means *files*. You can audit three operations on files: read, write, and execute. Objects are NOT associated with users. That is, audit records are generated whenever an audited object is referenced by any user (including *root*). The mechanism for object auditing generates pseudo-events, and the process works much like event auditing.

You can easily add additional files, including your application files, to the list of audit objects by extending the ***/etc/security/audit/objects*** file.

Information Collection

The audit logger (an AIX kernel function) constructs a complete *audit record*. This consists of the *audit header* containing standard information common to all audit records, and the *audit tail* containing data for a specific event. The audit logger appends every audit record to the *audit trail*. The audit trail can be written in one (or both) of two *modes*:

1. BIN mode causes the audit trail to be written into specified files. It can alternate between two files, in a somewhat complex manner.
2. STREAM mode causes the audit trail to be written to an in-memory circular buffer. A pseudo-device (/dev/audit) is provided to read from this buffer.

Using audit configuration parameters shipped with AIX (in ***/etc/security/audit/config***), the BIN mode alternates between ***/audit/bin1*** and ***/audit/bin2***. When one is full (as defined by the *binsize* parameter), the audit subsystem switches to the other BIN file to continue recording, and meanwhile adds the accumulated data in the first file (using the programs specified in ***/etc/security/audit/bincmds***) to ***/audit/trail***. After this is done, the first BIN file is considered empty and is ready for the next switch. When the audit subsystem is shut down, the data in the active BIN file is added to ***/audit/trail***.[63] The BIN method of output is batch oriented. You must shut down auditing to be certain that all audit records have been processed and added to ***/audit/trail***.[64]

The BIN data (which finally is written in ***/audit/trail***) is in an internal, somewhat compressed form. It must be processed by one of the audit programs (such as **auditpr**) to make it human-readable.

The STREAM process simply writes audit records in a circular buffer in memory. Events are overwritten as the buffer is overwritten. A pseudo-device (***/dev/audit***) is provided to read this circular buffer. You can write programs to read from this device. The default AIX configuration provides a program to do this. It reads the STREAM buffer and processes each record with the commands found in ***/etc/security/audit/streamcmds***. These commands (as distributed) format the output (for human readers) and write it in ***/audit/stream.out***. (This file is NOT cumulative; it is restarted every time the audit subsystem is restarted.)

The BIN mode offers safe output of the audit trail in a cumulative file. Because of buffering in the BIN files, you cannot accurately read BIN data in real time.

---

63 All the file names used can be changed by altering the config file. The file names used here are the ones used in the default config file.

64 The basic STREAM collection process (in the circular buffer) is more efficient than the BIN process. However, the standard overall STREAM process (using the setup shipped with AIX) is less efficient than the BIN process, because the backend processing for BIN is more efficient.

The STREAM mode can be read in real time by reading ***/audit/stream.out***, or by sending output directly to a display or printer.[65] You can use both STREAM and BIN modes at the same time, but this creates more overhead.

Audit Commands

There are five variations of the **audit** command:

- **audit start** is used to activate the audit subsystem. This is the only correct way to start auditing.
- **audit shutdown** ends the auditing subsystem, processing final BIN records (if needed) and removing the ***/audit/auditb*** file that is used as an "active" indicator by the audit modules.
- **audit off** is used to temporarily suspend auditing. Do not use **audit off** if you mean **audit shutdown**.
- **audit on** resumes auditing after an **audit off**.
- **audit query** displays the status of the auditing subsystem. In practice, you need to use **audit query | more** because the command typically generates 40-50 lines of output.

If you use these commands in the wrong order, or if the config file is not correct, the auditing subsystem can become confused. One indication of this is if the ***/audit/auditb*** file exists when you think the auditing subsystem is shut down. If the subsystem is confused, you can reset everything by deleting all the files in the ***/audit*** directory. (You should save anything useful in ***/audit/trail*** and the BIN files first.)

8.1 Audit Configuration

In AIX 4.1, audit subsystem controls are in the ***/etc/security/audit*** directory (including per-user controls). The ***/etc/security/audit/config*** file contains the key audit controls. It has these stanzas:

- *start*, which specifies whether BIN or STREAM (or both) should be used when audit is activated. By default, BIN is used.
- *bin* and *stream* contain controls for each mode. The names of the BIN files are specified here; the defaults are ***/audit/bin1, /audit/bin2***.
- *classes* defines several groups (classes) of audit events. Each class is a group of events that would logically be used together. The defined classes are: general, objects, SRC, kernel, files, SVIPC, mail, cron, and TCPIP. (You can define your own classes, using any of the possible audit events listed in the appendix.) Audit recording is enabled for classes. For example, you might audit the events defined as general and TCPIP for user *root*. (The class *objects* is a special case. It serves no purpose, and was left over from an earlier version of the audit subsystem. Do not use this class; you can delete it if you wish.)
- The *users* stanza lists specific users (and the audit classes assigned to them) to which event auditing applies. Data in this stanza is normally maintained through **smit** (using the *audit* parameter in user administration), but you can

[65] You can redefine the output file names, append to the output file, or use a totally different method of processing the audit records, by altering the ***/etc/security/audit/streamcmds*** file.

edit this file directly. The *users* stanza should always exist, even if no users are defined in it. If users are defined, they should have classes defined for each user. Do not have a user defined, without associated classes. An example of this stanza is:

```
users:
     root = general
     joe = general,files,cron
```

When auditing is started, it ALWAYS audits the *events* specified for every userid defined in the config file AND all the objects defined in ***/etc/security/audit/objects***. (To make object auditing work correctly, the objects should also be included in other control files; see the example below for details.) If you do not want object auditing, you should delete or rename the *objects* file. If you do not want event auditing, you should remove all the userids defined in the config file.

The ***/etc/security/audit*** directory contains several other files. The ***streamcmds*** file contains commands that are executed for STREAM audit records. As distributed, this file contains the command:

```
/usr/sbin/auditstream | auditpr > /audit/stream.out &
```

As you can see, this uses the program that reads the STREAM pseudo-device and pipes each record to the **auditpr** program. This program converts the record to human-readable form. The output is redirected to ***/audit/stream.out***. The ampersand causes this complete command to run in the background. (You can improve this command slightly by adding a *-v* flag after the **auditpr** command, to increase the amount of information in the output line.)

The ***bincmds*** file contains commands that are executed whenever a BIN file fills (or when the audit subsystem is shut down). In the distributed AIX system, this file contains:

```
/usr/sbin/auditcat -p -o $trail $bin
```

The environmental variables in this command are defined while the audit subsystem is running. You can change or add commands to either of these "commands" files. The most common addition is the **auditselect** command, which can be used to reject (or select) specific audit events, reducing the amount of audit information collected.[66]

The ***objects*** file lists all the objects to be audited whenever the audit subsystem is running. If this file exists and contains a list of objects, these are always audited -- this is independent of whichever userids are being audited. The file has a stanza for each target file:

```
/home/joe/good.stuff:
   r = "L_JOE_READ"
   w = "L_JOE_OUCH"
/payroll/input:
   w = "PAYROLL_WRITE"
```

You create appropriate event names, such as "L_JOE_READ." These event names are referenced in other files, as shown in "Basic Object Auditing" below.

[66] You should be as selective as possible when deciding on users, events, and objects for auditing. The **auditselect** command can reduce the amount of final audit output, but this is an "after the fact" filter that does not remove the overhead of collecting the audit information.

8.2 Basic Audit Usage

There are may ways to use the auditing subsystem. This section describes simple uses and can serve as a starting point for becoming familiar with auditing.

Basic BIN Auditing

Use **smit** to add audit classes for one or two users. You can use *root* as one of the users for these exercises.

```
smit
  Security and Users
    Users
      Change / Show Characteristics of a User

        *User NAME                        [joe]

         AUDIT classes                    [general,files]
```

Display ***/etc/security/audit/config***; you should see a user stanza for the userids you modified. Verify that the first stanzas in this file contain:

```
start:
       binmode = on
       streammode = off
bin:
       trail = /audit/trail
       bin1 = /audit/bin1
       bin2 = /audit/bin2
       binsize = 10240
       cmds = /etc/security/audit/bincmds
```

If you do not find these parameters, someone has altered your audit configuration and the following instructions may not work. (The following instructions assume you are *root*.)

Issue the command **audit start**. There is no response (meaning the command ended with code 0); you will simply receive the next prompt. Issue the command **audit query | more**. You should have a message that “auditing is on” and the PID number of the BIN process. You will also have listed all the defined events and objects.

Exercise the system, using the userid(s) you defined for auditing. Log into the system with this userid; do not **su** to it.[67] (If you are using a single-user workstation, it may be more convenient to audit *root*, since you must be logged in as *root* to issue the **audit** commands.) List files. List ***/etc/security/passwd***, which is one of the default files for object auditing. Try the **su** command (which causes one of the audit events listed in the *general* class).

Finally issue **audit shutdown**. Inspect the ***/audit*** directory. It should contain a bin1, bin2, and trail file. Use the command **auditpr -v < /audit/trail | more** to look at the audit trail. (If you used the **su** command, notice that the audit records contain your login userid, not the temporary su userid.)

[67] You can **su** to another user, but you need to understand the audit environment. Audit will always report your original login userid (and not the **su** target userid). However, it will audit only the events specified for the **su** target userid.

Basic STREAM Auditing

Change ***/etc/security/audit/config*** to enable STREAM mode.

```
start:
       binmode = on          (you can turn this off, if preferred)
       streammode = on
bin:
       trail = /audit/trail
       bin1 = /audit/bin1
       bin2 = /audit/bin2
       binsize = 10240
       cmds = /etc/security/audit/bincmds
stream:
       cmds = /etc/security/audit/streamcmds
```

Start the auditing subsystem again, with **audit start**. (Do not use **audit on**; this is used only after you have temporarily suspended auditing. If you attempt to start the auditing subsystem with **audit on**, it will confuse the system.) Display the ***/audit*** directory. The ***/audit/stream.out*** file should be growing as you use the system. Display data with **pg /audit/stream.out**. The STREAM file is in human-readable form, and can be listed with any editor or display program. Remember that, by default, the ***/audit/stream.out*** file is rewritten each time you start the auditing subsystem.

Basic Object Auditing

The above examples included object auditing of the objects defined in the default ***/etc/security/audit/objects*** file. We will expand this file in this example. Substitute your own file names for the ones shown here:

1. Edit ***/etc/security/audit/objects***, and add several stanzas such as:

   ```
   /home/joe/good.stuff:
      r = "L_JOE_READ"
      w = "L_JOE_OUCH"
   /payroll/input:
      w = "PAYROLL_WRITE"
   ```

   (Remember to include the colons after the file names.) Create any event names you wish, but select names that will not conflict with others in this file or the ***events*** file.

2. Edit ***/etc/security/audit/events***, and add several lines at the end of the file, after the other "object events" already defined there:

   ```
   L_JOE_READ = printf "%s"
   L_JOE_OUCH = printf "%s"
   PAYROLL_WRITE = printf "%s"
   ```

   These lines are for the **auditpr** program, to help it format output.

3. (There is no need to add anything to the *objects* line in the *classes* stanza of ***/etc/security/audit/config***. This *classes* line is not referenced by any audit function.)

Issue **audit start** using BIN or STREAM, as you prefer. (If you prefer STREAM mode, we suggest adding the *-v* flag to the **auditpr** command in ***streamcmds*** before starting auditing. This will cause the STREAM output to include more information.) Access the files you are auditing (such as /home/joe/good.stuff and /payroll/input in the examples). Inspect the STREAM output (or **audit shutdown** and inspect ***/audit/trail*** if you are using BIN mode) to verify that references to the objects were audited.

Minor Comments

The list of audit classes associated with a user is set when the user logs into the system (or is the target of an **su**) command. Changes to the audit classes of a user are not effective while he is logged into the system.

The **audit query** command may list more events types than you are currently using. Do not worry about it. (It lists all event types that have been used since the system was last booted.)

The audit processes will trace *thread ids*, which will be recorded in the header of all audit records. These are not printed by the **auditpr** command unless you specify that it be printed. (The **auditpr** and **auditselect** commands can include complex operands for selecting and formatting specific fields of audit records.)

The audit records produced by AIX 4.1 are not compatible with the records produced by earlier AIX systems. A conversion program is provided if you need to process older audit records with AIX 4.1.

Do not audit NFS or DFS files. The auditing subsystem provided with AIX 4.1 is not suitable for auditing distributed file systems. ;i1.audit userid

Audit always reports the login userid; this is not changed if the user switches to another UID with the **su** command. However, the selection of events to audit is controlled by the current (effective) userid (which can be changed with **su**).

8.3 Recommendations for Auditing

We do not recommend running the auditing function during your normal operations. The overhead can be substantial (depending on the events selected), and handling the output requires timely efforts. We do recommend that every system administrator learn how to use the auditing subsystem to accomplish basic functions. In particular, you should know how to start, stop, and display basic audit information. You should practice these steps until you can perform them without fumbling. You will need these functions if you suspect your system is being attacked.

You can add file names to the list of audit objects. You might decide on a set of critical production files and add these to the audit objects list. This causes no overhead unless you start the audit subsystem. Preparing a relevant objects list is best done before a critical situation arises.

An exception to these recommendations might be a critical server. You might want to define a minimal set of audit events and objects, and always run with audit active. In this situation, you are more likely to audit objects than users/events.

8.3.1 Audit Limitations

The limitations of the audit system should be understood before you consider building a major system monitoring function based on it. “Production” use of audit implies using the BIN mode. The BIN output files are *world writable*. The directory owning the BIN files is not accessible by *others*, but can be read/searched by the *system* group. A clever user (in the *system* group) could

introduce (or remove) audit events.[68] The BIN files are readable by anyone in the *system* group; this could have confidentiality implications.

Many audit events are defined (see Appendix C, "Audit Events" on page 115), but the "higher level" events depend on "higher level" programs to generate the events. For example, the **mkuser** command generates an audit event. However, it is possible to create a new user without using the **mkuser** command.

Many servers use database products (such as DB2) or on-line transaction processing monitors (such as CICS). These products do not interact well with the audit subsystem. For example, database monitors open their files, and all file access is through the database monitor. The audit subsystem is unaware of which user is requesting which data from the database monitor. The audit subsystem only sees the database monitor accessing its files.

These comments are not meant to degrade or discourage use of the audit system. It can be very effective, especially in commercial environments where *root* and *system* access is tightly controlled. Two styles of usage might be considered in this environment:

1. Monitor specific events and objects, and write a local program (or shell script) to routinely report usage.
2. Record many events and object accesses for use "after the fact" in investigating problems. This implies accumulating and managing significant amounts of data and will require some planning and effort to manage this data.

8.3.2 Auditing Products

We know of only one sophisticated product designed to use AIX audit data. This is *Stalker for AIX/6000*, from Haystack Labs, Inc, in Austin, Texas.[69] This product will accept audit data (from multiple AIX systems) and permit a central administrator to browse it, with many selection criteria. The unique function of *Stalker* is a *misuse detector*. This is a program that works with a large database (furnished by Haystack) of *misuse signatures* (series of audit events) that are characteristic of specific attacks or problems in the system. Detected problems (or potential problems) are reported to the administrator.

Stalker can work with a variable set of audited events and objects. The more events/objects audited, the more useful the program. The downside is that more audit recording produces more system overhead.

[68] Of course, the *system* group should contain only trusted administrators. The number and nature of users in this group is one factor in determining the extent of reliance on audit output. The "world writable" permissions of the BIN files may change in future releases of AIX.

[69] The telephone number is (512)-918-3555.

Chapter 9. Other Topics

This chapter briefly discusses several topics related to AIX system security that are not covered elsewhere in this document.

9.1 Firewalls

A "firewall" is a system that protects your network from unwanted interactions with an external network. The most typical use is to connect your network with the Internet. A firewall can be a function on a shared system, but it usually is a small, separate, dedicated system. Software to create a firewall is available from several sources, including "free" anonymous FTP servers. The most common approach is to create a "wrapper" that intercepts programs run through **inetd**.

At the time this was written, IBM was beta testing (in a number of customers' installations) a product named *IBM NetSP / Secure Network Gateway for AIX 6000*. This is a firewall-type product. The version being tested is only for AIX 3.2.5, and (at the time this was written) had not been tested with AIX 4.1. The requirements for the program are an AIX system with two (or more) LAN adapters (token ring, ethernet, SLIP), 32MB memory, 800MB disk, and a graphics console.

This product provides rewritten TCP/IP functional modules instead of using the "wrapper" approach.[70] The initial product provides gateways for **telnet** and **ftp**; gateways for other services (such as **gopher** and **mosaic**) may be added later. Users in the protected network log into the Firegate system to use the gateways.

The product also supports the *SOCKS* protocol. This is a (new) standard protocol for clients in a protected network to access servers in an unprotected network. It requires rewritten client software, such as new **telent** and **ftp** modules (and several are included with the product). A "socksified" client uses the protected gate automatically; the user appears to connect directly to the external server. In other words, the use of a firewall-type system becomes transparent. Over time, most TCP/IP packages will provide and support "socksified" functions.

Administration is through a graphical interface. The administrator has a fine level of control over users and functions that may pass through the gateways.

If your organization's production networks are connected to the external world (usually meaning the Internet), you should strongly consider installing a firewall-type interface. The number and variety of system attacks over the Internet cannot be ignored.

[70] This is a more secure and better architected design.

© Copyright IBM Corp. 1994

9.2 X Windows

A full discussion of security for X Windows is a complex topic and beyond the scope of this document. The following material may help with initial security planning.

The terminology for X Windows must be understood. A *server* is the unit with the display; it is a *display server*. A *client* is the system with the computational program that is sending output to (or reading input from) the server.

Basic security begins with the display server. It can control which client systems may use the display server. There are two methods for doing this:

1. The **xhost** command may be used to add or delete systems from the list of permitted client systems. This command, in effect, makes a temporary list (which does not survive rebooting).
2. You can create files named ***/etc/X**n**.hosts***, where n is 0, 1, 2, ... (the number of the logical display) containing lists of the client systems permitted to connect to this display server.

You can look for, and control, ***/etc/X?.hosts*** files on your system. Unfortunately, you cannot readily find user scripts with **xhost** commands authorizing large numbers of client systems connections with an X Windows server on your system. Some users make shell scripts containing **xhost** commands for every client they might ever want to use, and this is certainly an exposure.

A fundamental problem with X Windows security is that, once a server permits a client system to connect to it, any program running in that client can access (and, to a large extent control) the display server. Newer protocols and the new XDM control processes address aspects of this problem, but these controls are not generally understood or used. Fortunately, the general exposures of X Windows usually apply to workstations (with graphics terminals) instead of servers (which, by some definitions, do not have graphics terminals).

Part of the customization for XDM (and, potentially for other elements of X Windows) *on display servers* permits the specification of files that will be automatically executed *as root* when the server is started or stopped. This is a potential security exposure. The standard directories for customizing X Windows are normally protected, but it is possible to override many parameters.

XDM (a new X Windows manager in AIX 4.1) cannot readily be stopped once it is started, and it might be automatically started (if you select to do so). This could be considered a security "extra," because it prevents basic line-mode access to the terminals it controls. This aspect might be relevant for workstations in some exposed environments.

9.3 The skulker Script

AIX is delivered with the file ***/usr/sbin/skulker***, commonly known as **skulker**. This is a shell script that deletes a variety of (presumably) unwanted files. It is a moderately complex shell script; parts are easy to read and parts are rather difficult. As delivered, it is suitable for local file systems. According to the comments, you may need to edit it and set *NATIVE=/native/* if used with distributed systems.

You can execute **skulker** from the command line (if you are *root*), or you can execute it from **cron**, which is considered the "normal" use. It is not automatically executed by **cron** in the base AIX system. There is a line in ***/var/spool/cron/crontabs/root*** for it, but the line is a comment. You can edit this file and uncomment the line if you want to have **cron** run **skulker** every night.

The following files are deleted by **skulker**:

- spooled output files older than four days
- files left in the mail queue more than two days
- ordinary files in ***/tmp*** that are more than one day old
- ordinary files in ***/var/tmp*** that are more than one day old
- *.bak, .*.bak, a.out, core, proof, galley, ...*, and ed.hup files (with a few restrictions) that are more than one day old
- .putdir directories more than one day old

You may change **skulker** as you wish, but be careful with it. It is executed with *root* authority, and any changes should be well tested.

9.4 Controlling cron and at

You should read the AIX documentation for the **crontab** command before using **cron** functions. While it is possible to edit some **cron** files directly, the **crontab** command provides the normal method of adding, changing, and deleting **cron** jobs. The AIX **cron** system (through the **crontab** command) builds separate files for jobs from different users, and has removed the many *root* exposures that were well known on older UNIX systems.

You should consider whether any of your normal users should be permitted to submit **cron** or **at** jobs. On earlier systems, there was a common need for these functions because there was not sufficient processor power to perform many jobs on-line (in the "foreground"). This has changed on more modern systems. Today, the most common use of these functions is to run regularly scheduled production jobs. You may not want your normal users introducing any **cron** or **at** jobs.

There are two ways to control the use of these functions. You can prohibit specified users from using these functions, or you can limit usage to a list of specified users. This involves four files:

```
/var/adm/cron/cron.allow
/var/adm/cron/cron.deny
/var/adm/cron/at.allow
/var/adm/cron/at.deny
```

The two "deny" files exist in the distributed system, but are empty. The two "allow" files do not exist in the distributed system. If they exist, the "allow" files take precedence, and the "deny" files are ignored. If an "allow" file exists, only the users listed (by userid) in the file are permitted to use the function. This applies even to *root*; that is, *root* must be listed in an "allow" file in order to use the associated function.

Especially for larger servers, we suggest you create ***/var/adm/cron/cron.allow*** and ***/var/adm/cron/at.allow*** files containing the name *root* and the names of

whoever is expected to establish scheduled production jobs. The files contain the simple userids, one per line.

The **cronadm** command is useful for inspecting current **cron** and **at** information:

```
cronadm cron -l                    (list all cron files)
cronadm cron -l joe                (list joe's cron files)
cronadm cron -v                    (list job submission status)
cronadm at -l                      (list existing at jobs)
cronadm at -l joe                  (list joe's at jobs)
cronadm at -v                      (list submission status)
```

The command can also be used to remove jobs from these queues. If a cron (or at) job does not exist for a specified user, you will receive an AIX message about "file or directory not found."

Chapter 10. Checklists and Reviews

The following lists may be useful for initial installation, routine operation, or a review of AIX security. No lists such as these can be complete, or can be simple cookbooks. Understanding and perseverance is needed for these activities; checklists are just an additional tool.

10.1 Planning

As delivered, AIX is a totally "open system." It has no effective security, but provides the tools for the administrator to create a secure system. It is up to the administrator to establish initial security and maintain security through routine administrative actions.

You should obtain or produce a formal security policy (no matter how simple it may seem). This provides a target for your security implementation. Sometimes, in the midst of too many details, the target becomes confused -- especially if many people are involved.

You should, at least initially, consider separately the security aspects of:

1. Your base AIX systems,
2. Your network environment, and
3. Your NFS (and NIS, if used) environment.

Attempting to consider all these at the same time can be confusing. Different goals and different risk factors/assessments apply to these different areas.

10.1.1 Initial Installation

- Install the TCB. (This may be an installation option on some versions of AIX 4.1).
- Set a password for *root* as soon as your system is usable.
- Set the following password restrictions in the default stanza of ***/etc/security/user***.

  ```
  pw_restrictions:
          maxage = 12              (force change after 12 weeks)
          maxrepeat = 3            (max three repeated characters)
          minalpha = 1             (at least 1 alpha character)
          mindiff = 3              (at least 3 different from last time)
          minother = 1             (at least 1 nonalpha character)
          maxexpired = 4           (allow logon 4 weeks after expired)
          histexpire = 26          (prohibit reuse for 26 weeks)
          histsize = 8             (prohibit reusing last 8 passwords)
          pwdwarntime = 14         (start warning 14 days before expire)
  ```

- Define a timeout value. Place it in ***/etc/profile*** if it is the same value for all users.

  ```
      TMOUT=1800              (for Korn shell)
      TIMEOUT=1800            (for Borne shell)
      export TIMEOUT TMOUT
  ```

  The timeout value is expressed in seconds. For example, the value 1800 means the shell should timeout (exit) if there is no activity for 30 minutes.

© Copyright IBM Corp. 1994

Set both TMOUT and TIMEOUT if your users might use either shell. See 3.7.1, “Additional Authentication Methods” on page 33 for more information.

- Update the shell prompt, as discussed in 3.4.2, “Prompts” on page 26, if appropriate. This is not a direct security issue, but a shell prompt that indicates the current directory helps prevent errors by new and old users alike. ***/etc/security/.profile*** is a good place for it.
- Redirect output of **skulker** and similar reports to a single file, for example, ***/tmp/dailyreport*** - this makes it easier to monitor system activities and status daily.
- The **securetcpip** command disables various daemon services. Use this command (once) if **rlogin** and related commands are not required.
- In the ***/var/adm/cron*** directory, use ***cron.allow***, ***cron.deny***, ***at.allow***, and ***at.deny*** files to control access to these functions. See 9.4, “Controlling cron and at” on page 93.
- Change the login herald to identify your system. You may want to make a distinctive display, making your system herald easily recognizable. See ***login.cfg*** in 3.7, “Files Associated With User Accounts” on page 30.
- Learn how to change the message of the day.
- Run **tcbck** to establish a base; print the configuration file. Review and fix any reported problems now. See 7.2, “Using the tcbck Command” on page 78.
- Assign different *root* passwords to different machines. The administrator should ensure that distinct *root* passwords are assigned to different machines. You may allow normal users to have the same passwords on different machines, but never do this for *root*.
- Have emergency procedures. In the event the administrator cannot be reached during an emergency, an authorized person should have access to the necessary password. Use of this procedure should be logged and the password changed immediately after use. This is simply good business sense.
- Consider disabling all remote and dial-in terminals at the end of the day. Enable them in the morning. This can be done with the controls in ***/etc/security/login.cfg***.
- Are other port controls needed? Should certain users be restricted to (or from) certain ports?
- Carefully review all the parameters in the *default* stanza of ***/etc/security/user***. Set appropriate defaults before you create users, so you will not need to specify many parameters for each new user.
- Consider disabling login capability for *root* on any system where more than one person knows the *root* password. This forces users to login under their own userid and then **su** to *root*. Accounting and/or ***/var/adm/sulog*** will have a record of these users. (Of course, a *root* user can alter these records, but this implies intended improper activities.)
- Edit ***mkuser.default***, as discussed in 3.3, “Users” on page 16.
- Consider enabling SAK for all terminals, and allowing all users to use the trusted shell (if they want to use it). See Chapter 7, “Trusted Computing Base” on page 77.

10.1.2 Continuing Activities

- The *root* password should be changed on an unannounced schedule by the system administrator. If multiple people need to know the password (for a large server with 24-hour operation, for example), be certain the new password is communicated in a reliable way.
- Take backups. Keeps records. Be certain someone else knows how to locate the most recent backup and recover files from it. This requirement applies to servers and workstations.
- Beware of shell archive files (*shar* files). These are a common way to distribute "freeware" or "shareware" or to obtain files from anonymous ftp servers. Never execute a *shar* file while operating as *root* unless you have examined the whole file and trust it. This can be difficult. A user (or even your management) may approach you with a wonderful program that is needed immediately. It was obtained from "somewhere" (not a very trusted source) by various file transfers (not a trusted channel) and needs to be installed[71] by *root*. This situation must be handled with considerable care and tact.
- A new user, created via **smit**, is not able to use the system until his password is created (with **smit** or with the **passwd** command). You can create new users before they are needed and delay creating their passwords until the users require access to the system. A new user is required (by AIX) to change his password the first time he logs into the system. If you (the administrator) assign "standard" passwords when a new user is created, you have an exposure. After the new account is created, anyone could be the first logged in user and thus set a new password. You should create unique and obscure initial passwords for new users, OR require the new user to log in immediately after you create his initial password. (When he logs into the system, he will be asked to change the password.)

  When adding a new user:

  1. Be certain the user understands how to make an acceptable password, and changes his initial password. Even unshared systems should have good passwords.
  2. Explain your policy about unattended terminals and timeout operation.
  3. Give a written copy of your organization's security policy to the new user.
  4. Ask the new user to login. The system will ask him to change his password. Be certain he does this.
  5. Be certain the user knows where (in ***/u/userid***) to keep his files - and where not to keep his good files (such as in ***/tmp***).
  6. Instruct him in the dangers of revealing (or "loaning") his password to anyone.

- Do not allow users to share a userid (by sharing the password) or a UID (by equating several accounts to the same UID). There are exceptions to this, but these must be carefully considered.
- When assisting a user, do not **su** to *root* from his session. If you do this, you are using his environment (with his *PATH*) and this opens a large number of exposures. If you must do this, then use full path names for all commands you use while executing as *root*.

[71] A *shar* file is "installed" by executing it.

- Beware of a user who changes *IFS* (input field separator) in his profile. Do not allow it to be changed in ***/etc/profile***. A knowledgeable user can play many clever terminal tricks with *IFS* and cause endless trouble.
- Do not place the current directory in the *PATH* for *root*. Do not allow it to be specified this way in ***/etc/profile***. The default PATH for AIX, which also applies to *root*, has the current directory as the last element in the PATH. You must create a ***.profile*** for *root* to override the default PATH in the default profile (in ***/etc/profile***). Typically, *root*'s home directory is ***/***, although you can assign another home directory. A ***.profile*** is in a user's home directory.
- A umask value should be set for users. With AIX 4.1, a umask can be specified as part of the **smit** panel for adding or changing a user. The normal umask value is 022, although 027 (disables any "world" access) may be better in some cases. Specific umask values may be placed in individual ***$HOME/.profile*** files. (Remember that a user can change his own umask value at any time. The administrator cannot easily prevent this, although it can be made slightly more difficult by placing *alias* statements in the user's ***.profile***)
- Always ensure that the permissions that are set for exported NFS directories are as limited as possible. For example, if you wish to export ***/usr/lpp/appl1/bin*** and ***/usr/lpp/appl2/bin***, you should have two entries in ***/etc/exports***:

  ```
  /usr/lpp/appl1/bin
  /usr/lpp/appl2/bin
  ```

  You should not have the higher level directory entry:

  ```
  /usr/lpp
  ```

  Do not export "high-level" directories unless absolutely necessary. For example, do not export ***/u*** if all you really need to export is ***/u/johnj/dbdir***.
- Never routinely operate as *root*. Login to your normal userid and use the **su** command to become *root* only for necessary steps.
- If you have activated any auditing functions you should check their output at least daily, looking for unusual events. Unusual events may include activity at odd hours, repeated login failures, repeated failures with the **su** command, and so forth. Possibly delete the audit output when finished; it tends to grow rapidly.
- Consider adding **tcbck** to **cron**. (This might be done a little later, after your system is operational). See 7.1, "TCB Description" on page 78.
- Use **tcbck** daily or at least weekly. See 7.2, "Using the tcbck Command" on page 78.
- Update the tcbck profile when important files (from a security viewpoint) or *suid* programs are added to the system. See 7.2, "Using the tcbck Command" on page 78.
- If you use the accounting system, inspect the output on a fixed schedule.
- Inspect ***/tmp/dailyreport*** daily, if it exists.
- Run **errpt** at reasonable intervals. See 4.5, "AIX Version 4 Error Logging" on page 57.
- Display ***/var/adm/sulog*** and look for unusual patterns. Delete it when it becomes too large to easily scan, or delete it every time you inspect it.

- Check **at** and **cron** jobs, especially after someone leaves the organization. This can be done (by *root*) with **cronadm at -l** *userid* and **cronadm cron -l***userid*.

10.2 Reviewing a System

The following considerations and tests may be useful for reviewing the security environment of an AIX system.

- Are groups used effectively?
- Is there a written security policy (no matter how simple it may be)?
- Who takes backups? This question is much more significant for servers than for workstations, but it is important for both. Do several people know where backup records are kept? Can several people find the most recent backup and restore files from it?
- Note that a user account can be disabled in several ways:

  1. The password field in ***/etc/passwd*** can contain an asterisk.
  2. The password field in ***/etc/security/passwd*** can contain an asterisk.
  3. The expires field in ***/etc/security/user*** can contain an expired date. The date 0101000070 (JAN 1 1970) is normally used for this purpose.

  AIX tends to use a mixture of the second and third methods. When verifying that an account is disabled, you should check in this order.

- All active user accounts should require passwords. Any account without a password should be deactivated. Use the command **pg /etc/passwd**, and verify that the second field of each line is not null. (You can do the following check at the same time.) (The **usrck** command performs this same check, but you should scan ***/etc/passwd*** at least once anyway.)
- The ***/etc/passwd*** file should not contain any passwords. You can list the file (with the **pg /etc/passwd.** command) and scan for encrypted passwords (which would be the second field in each line). The second field should contain an exclamation point, indicating that the encrypted password is stored in ***/etc/security/passwd***.
- Is the *root* password documented, for emergency use?
- Do administrators have a login and a password different from that of *root*?
- Display ***/var/adm/sulog*** and look for unusual patterns.
- Run **pwdck -n ALL**. This command looks for irregular contents in ***/etc/passwd***, ***/etc/security/passwd***, and ***/etc/security/name*** files. Problems are listed on the screen (which you can redirect to a file). There is an option to automatically "fix" any problems found. We suggest you do not use this option except under unusual circumstances. If you manually fix problems, you will be more aware of their origin and other side effects not reported by the "ck" commands.
- Run **grpck -n ALL**. This command is similar to the **grpck** command. It checks relationships between the ***/etc/group, /etc/passwd, /etc/security/passwd, /etc/passwd.pag,***, and ***/etc/passwd.dir*** files. (The last two files listed are for a small database to allow AIX to resolve UIDs and GIDs quickly.)
- Run **usrck -n ALL**. This command is similar to the **pwdck** command. It checks relationships between the ***/etc/passwd***, ***/etc/security/user***,

/etc/security/limits, ***/etc/security/passwd***, and ***/etc/group*** files. You may receive messages that several system-owned accounts have expired. You can ignore these messages.

- Are TMOUT and/or TIMEOUT environmental variables set? Use the **set** or **env** command to display current environmental variables for the account you are using. These variables are often set in ***/etc/profile***, and you can **pg /etc/profile** to check this.
- Is *root* enabled for login? Is it intended to be enabled for login? Use **smit** to display the characteristics of *root*.
- Log into the system as *root* (or **su -** if login is not permitted). Use the **env** command to display the PATH variable. Is the current directory in this PATH? If so, is it at the end of the PATH? (It is best if the current directory is not in *root*'s PATH. If it is in the PATH, it should be listed after the system directories.)
- The default permissions for a $HOME directory (when **smit** is used to create a new user) are 755. This allows anyone to display the user's $HOME directory; this may, or may not, be appropriate for your installation. Use the command **ls -l /home** to display the permissions for all users' $HOME directories (assuming you are using the normal AIX directory structure).

  HOME directory permissions 710 prevents anyone except the owner of the directory from writing in it. It also prevents groups and others from visiting a user's HOME directory, searching for files with insecure permissions or interesting information.

  A permission mode of 711 on a user's HOME directory allows others to traverse to lower directories within the HOME directory which have more open permissions, but not to display the contents of the base $HOME directory. Files that require access by others should be placed in lower directories.
- Initialization files, such as ***$HOME/.profile*** , should not be writable by anyone other than the owner. The permission modes for these files should be 640 or 600. If writable by others, they could be modified in a way to compromise all files to which the user has write access, or the user's *PATH* could be changed. This is a critical check. Try the command:

  ```
  find /home -name .profile -exec ls -l {} \;
  ```
- The default permission mode (a user's **umask**) should be set to 022 or 027. The default **umask** can be found in ***/etc/security/user***, and can be set through **smit**. Is a default umask value set for the current user? Enter the ***umask*** without operands.
- The user's path should be reviewed to ensure that system directories are checked before local directories. Using a "normal" userid (not *root*) issue an **env** command and examine the displayed PATH. Issue an **ls -a** while in this userid's home directory. Determine if there is a ***.profile*** containing a PATH that overrides the default path.
- While logged in as *root*, issue the command **lsuser** *userid* where *userid* is any defined userid that appears interesting. This command will display almost all the security-relevant administrative settings for that user. You can use the command **lsuser -f ALL >> /tmp/ulist** to obtain the same listing for all defined users, directing the output to a temporary file (because it will be too large to display directly). You can print this file and take it away for desk checking.

- ***/etc/hosts.equiv***, ***.rhosts***, ***.netrc*** - Are these files permitted by your security policy? To check for these files, use:

```
find /home -name .rhosts -print
find /home -name .netrc  -print
pg /etc/hosts.equiv
```

- ***/etc/hosts*** - Check to ensure that no new host systems have been added and that the IP addresses are correct.
- ***/etc/inetd.conf*** - This file defines which TCP/IP services are enabled. Check to see that unwanted services have not been enabled on your system. A line connecting a program to an obscure port address could provide a path into your system for a *root* user. List the file with **pg /etc/inetd.conf**.
- **netstat** provides network status information. It is commonly used when diagnosing network problems. It can also provide information that is useful to check network security. For example:

  ```
  netstat -p tcp
  ```

  provides information about the TCP/IP protocol since booting the system. Look for things such as failed connection attempts. This may mean someone is trying to break into your host.
- If the **securetcpip** command was used (during installation, for example), verify that the unwanted commands have not been reenabled. For example, the command **ls -l /usr/bin/rlogin** should show --------- for the permissions field.
- Review hidden files. Some users think they can hide various misdeeds in them. You may also uncover large amounts of wasted disk space, especially if INed is used. The command **find / -name .??* -exec ls -l {}\;** can be used, although you might want to redirect output to a file. Expect to see .profile and similar names.
- The system does not allow rebooting with a CD-ROM or tape if the key switch is in the Normal or Secure position. Is it routinely left in ″Normal″ or ″Secure″ and the key removed? Who has the key? Backup for this person?
- The *root* password should be changed on an unannounced schedule by the system administrator. Who is the backup person? Is there a reliable method of informing him of a new password?
- Are portable file systems used? Are these *always* mounted with the *nosuid* option? Enter the **mount** (with no operands) to see the options for currently mounted file systems. Display ***/etc/filesystems*** to examine defined filesystems. A *root* user can mount filesystems not in this list. The *nosuid* option apparently cannot be specified in ***/etc/filesystems***.
- Is a CD-ROM drive attached to the system? If so, who can **mount** CD-ROMs? Are CD-ROMs always mounted with *options = nosuid*? Files can be executed from a CD-ROM, even with *suid root*, making CD-ROMs a potential major hole in system security. (Older UNIX systems that supported file systems on diskettes have the same exposure for mounted diskette files.)
- System directories should have permission mode at least as restrictive as 755. Examine all ″world writable″ directories. For each of these, check whether the sticky bit is set. A sample command is:

  ```
  find / -perm -002 -type d -exec ls -ld {} \;
  ```

  World writable directories owned by *root* or *bin*, especially if not protected by the sticky bit, should be examined with some care. (You should be *root* to

use the above command without generating many error messages. This **find** command will examine the entire directory tree, included NFS mounts and CD-ROM data. You might want to unmount these before spending much time with **find**).

- User home directories that are world writable are common and are exposures. You can find these with:

```
find /home -perm -002 -type d -exec ls -ld {} \;
```

  There are few excuses for a user's home directory to be world writable. Attacks through such directories are a basic tool of system intruders.

- Is the **tcbck** facility used? That is, is the command used regularly to (1) verify the current TCB, and (2) to update the tcbck database with changes to the TCB? Try the commands:

```
tcbck -n ALL
tcbck -n tree
```

- Secure sensitive system files, such as UUCP files, cron tables, system log files, and system source code. These files should be closed to all users.
- Review permissions on device files. Insecure permissions on device files allows direct access to hardware devices and their kernel data structures. Verify these values:

```
666 for any tty device not in use
664 for any tty device in use
660 for hdn (disk) devices
600 for console
440 for mem and kmem
664 for kbd
```

  These are the settings in a standard AIX 4.1 system. You may want more restrictive settings.

- All device files should reside in ***/dev***. Any device files residing outside ***/dev*** should be investigated.
- Check for unowned files. The command **find / -nouser -print** can be used.
- Check permissions on the directories and files containing the **cron** controls. These are in the path ***/var/spool/cron/conntabs***. Only *root* and *cron* should have access to most of these files. Since most programs run by **cron** execute their commands as *root*, access to ***crontab*** can allow an intruder to search for programs that are writable by others and then insert code to obtain *root* access and compromise the system. (One directory in this path, ***/var/spool/secretmail*** is world-writable.)
- Look in the ***/var/adm/cron*** directory. Do the ***cron.allow***, ***cron.deny***, ***at.allow***, or ***at.deny*** files exist? Some combination of these files should be used to control these functions. See 9.4, "Controlling cron and at" on page 93.
- Issue the command **cronadm cron -l** to list all current **cron** jobs. Inspect the listing for any obvious problems. Do the same with the **cronadm at -l** command.
- Is password aging used? Review the ***/etc/security/user*** file.
- Review the ID/password file for unused accounts. Unused logins should be voided or removed to prevent intruders from gaining unauthorized access. Disgruntled employees may give passwords away upon leaving the organization.

- An inventory of all *setuid/setgid* programs should be obtained. The **tcbck** can do this. The system administrator should review all *setuid/setgid* programs owned by *root, bin,* or *daemon*, or owned by the groups: *bin, kmem,* or *mail*. The initial list of *setuid/setgid* programs should include the owner, group, permission, and checksums. This list should then be compared with ongoing lists, and any unrecognized *setuid/setgid* programs should be disabled and reviewed. (This task is easier to specify than to do. If ***/etc/security/sysck.cfg*** exists (and has been maintained), you can execute **tcbck** with the *tree* parameter. This will search for unknown *suid* files. If **tcbck** cannot be used, you will need to use **find**, and perform much manual verification.)
- *setuid/setgid* programs should have access permissions of 511 (or similar) to insure that others cannot examine the programs.
- When reviewing *suid* programs, you must review both the program name and the complete directory path containing the program. Program names need not be unique in UNIX. A user can have a program named “passwd,” for example. It is the directory path (and the current PATH environment) that prevents this program from being executed instead of the “real” **passwd** program.
- Check for **at** jobs with owner *root* on a regular basis, and immediately if someone with knowledge of the *root* password leaves the organization. The **cronadm at -l** command can be used.
- Check for users who change *IFS*. (This is a low-priority check. A user experimenting with IFS is most likely to change it dynamically, rather than through entries in a profile.)
- Under some circumstances, the following commands bypass normal authentication controls:
  - **rlogin**
  - **rcp**
  - **rsh** (**remsh**)

  If you must use these commands, do not use ***.rhosts*** or ***/etc/hosts.equiv*** files to automate their operations. Use **find /home -name .rhosts -print** to determine which users have .rhosts files.
- **tftp** has no user authentication. It should not be used in any network where any security is required. To disable it, run **securetcpip** or do the following:
  1. Comment out the line relating to **tftpd** in ***/etc/services***.
  2. Comment out the line relating to **tftpd** in ***/etc/inetd.conf***.
  3. Either remove the **tftp** command from the system or change the permissions on it to prevent use. The command is ***/usr/ucb/tftp*** .
- If you are using the system in a secure mode, you should not allow **ftp** and **rexec** to use ***.netrc*** for automatic login.

Appendix A. DoD Classes

The "Orange Book" criteria came from a task force of the Defense Science Board started in 1967. The original report, *Security Controls for Computer Systems* was published in 1970. The current document is *Trusted Computer System Evaluation* (DoD85). This material is important for many installations, but it is often misunderstood and misused.

There are two quite distinct sets of criteria. One set defines a number of security features. The other set defines the tools, information, and some of the processes required to verify the correctness (design and operation) of the features. The security features are the *Security Policy* and the second set is the *Assurance* criteria.

Seven security levels are defined. These levels are:

- D - Minimal Protection
- C1 - Discretionary Security Protection
- C2 - Controlled Access Protection
- B1 - Labeled Security Protection, Mandatory Access Control
- B2 - Structured Protection
- B3 - Security Domains
- A1 - Verified Design.

and the following terms have specific meanings:

- *Security Policy*. These are the "rules" that the security features enforce. For example: every user must have a password, or only a file owner can change the access list for the file. The specific rules will vary in different security levels and in local installation standards. The total set of rules enforced by the system forms the security policy of the system.
- *Identification and Authentication (I&A)*. Subjects must be uniquely defined. A subject is a user or a process.
- *Marking* or *Labeling*. Objects (usually a file) must be associated with a security label that contains a security level and security category. For example, "Secret" is a level and "Research" might be a category.
- *Accountability*. This refers to complete and secure records of actions that affect security. Such actions include user setup, assignment or change of security levels, and denied access attempts.
- *Assurance*. This refers to system mechanisms that enforce security; it must be possible to measure the effectiveness of these mechanisms.
- *Continuous Protection*. The hardware and software mechanisms that implement security must be protected against unauthorized change.
- *Object Reuse*. This refers to memory blocks or disk blocks, for example. A program should not find "left over" data from another process or file when it acquires a memory or disk block.
- *Covert Channels*. This refers to indirect means of delivering information to an unauthorized user. For example, a program might make subtle changes to unclassified messages to convey classified information, or leave data in a shared memory location.

© Copyright IBM Corp. 1994

The following table (taken from DoD85) contains the requirements for the various security classes.

Criteria	D	C1	C2	B1	B2	B3	A1
SECURITY POLICY:							
Discretionary Access Control	x	R	R	-	-	R	-
Object Reuse	x	x	R	-	-	-	-
Labels	x	x	x	R	R	-	-
Label Integrity	x	x	x	R	-	-	-
Exportation of Labeled Information	x	x	x	R	-	-	-
Labeling Human-Readable Output	x	x	x	R	-	-	-
Mandatory Access Control	x	x	x	R	R	-	-
Subject Sensitivity Labels	x	x	x	x	R	-	-
Device Labels	x	x	x	x	R	-	-
ACCOUNTABILITY:							
Identification and Authentication	x	R	R	R	-	-	-
Audit	x	x	R	R	R	R	-
Trusted Path	x	x	x	x	R	R	-
ASSURANCE:							
System Architecture	x	R	R	R	R	R	-
System Integrity	x	R	-	-	-	-	-
Security Testing	x	R	R	R	R	R	R
Design Specification / Verification	x	x	x	R	R	R	R
Covert Channel Analysis	x	x	x	x	R	R	R
Trust Facility Management	x	x	x	x	R	R	-
Trust Recovery	x	x	x	x	x	R	-
Trusted Distribution	x	x	x	x	x	x	R
DOCUMENTATION:							
Security Features User′s Guide	x	R	-	-	-	-	-
Trusted Facility Manual	x	R	R	R	R	R	-
Test Documentation	x	R	-	-	R	-	R
Design Documentation	x	R	-	R	R	R	R

An ″x″ means no requirement. An ″R″ means this class has additional requirements over the lower classes. A ″-″ means this class has the same requirements as the next lower class.

As can be seen, the requirements listed in this table are very general. Many systems can claim to cover various levels of these requirements. To have any real meaning, **a system must be certified for a particular level**. This means that the system was examined (in great detail) by a U.S. government agency and **certified** to operate at a certain security level. This certification is a long process and can be expensive for the system developer. The specific tests and criteria are designed for national security installations and may not be completely appropriate for commercial users. Level ″D″ sometimes is applied to a system that failed tests for a higher level.

Certification does not provide a guarantee or warranty that the security system is perfect. It merely says that it satisfied the agency performing the tests. However, these tests are generally accepted to be rigorous.

Discretionary Access Control (DAC) and Mandatory Access Control (MAC) are very important concepts. DAC allows the owner of a file to set the security parameters for the file. In AIX this is the owner setting permission bits or ACL

controls. MAC means that the system (through control parameters set by the security officer) automatically controls the security parameters of a file. The owner of the file cannot change these. AIX does not support MAC.

A system can have both MAC and DAC. The security officer (using various control lists) decides which files (or categories of files) are controlled through MAC and which are allowed for DAC. The DoD standard does not specify any particular implementation for these facilities, and different systems use very different mechanisms to implements these controls.

Human-readable output labels, specified in the DoD table, means the system must automatically print security labels on output. This is independent of any particular application program. This can be a difficult requirement because the operating system does not understand what an application program is printing on any particular page. If the system overprints ″TOP SECRET″ at some page location it may overlay important application output. (The MVS/RACF solution uses only laser page printers, and prints the security label in a nondestructive manner.)

A.1.1 Levels for Commercial Users

Class C is the most important security level for most commercial installations. (Almost all systems at this level control Object Reuse and provide an Audit facility, although these are C2 requirements only.) Class C is important for two key reasons, neither of which is directly related to the specific security features of C1 or C2:

1. A reasonable set of security features are included. This set is sufficient to provide reasonable control for most installations without creating a major administrative burden.
2. The system has been independently tested and certified to perform these functions correctly.

Unless there is a specific need for higher security, C1 or C2 should meet generally accepted security practice requirements ---- if used intelligently. From the user′s point of view, there is little difference between C1 and C2.[72] The assurance and testing requirements for C2 are more rigorous and, in this sense, a C2 system is better than a C1 system.

From a normal commercial viewpoint, class B introduces two major changes: mandatory security access (MAC) and security labels. Security labels include both security level (secret,...) and category (research, payroll, ...). A user might have access to secret data in research, but only unclassified data in payroll. The implementation of security labels requires substantial changes to most systems. Both security labels and MAC may require considerable administrative effort to implement and maintain.

The B2 and B3 classes become very rigorous and are difficult to certify. Class A security would be very unusual for a commercial installation.

---

[72] C2 requires an Auditing feature and control over Object Reuse. In practice both of these are already in C1 systems.

A.1.2 Comments

System owners often assume the following:

- A higher security class is better.
- A higher class will take care of security needs with less effort.
- Certain applications need a higher security class.

These assumptions are all false and lead to misuse and misunderstanding of the security classes.

One key factor is the sharing of systems. For example, consider a customer-account system in a bank. This appears a good candidate for a very high security level. However if this system is implemented as a totally unshared system (that is, there are no users other than this application), and there are no ″timesharing″ terminals attached to the system, and physical security is sufficient, and so forth, then a formal security class is pointless. Likewise a small departmental system shared by users of the same ″security level″ (whatever this may mean in a given organization) is unlikely to need as much protection as a large system shared by many varied users of different security levels and categories.

More security functions usually require more administrative effort. A higher class does not automatically provide more security. It is *capable* of providing more security, with proper administration. Conversely, a higher class system may work very poorly and eventually become unusable if the security functions are not properly administered. **Do not buy or install more security than you are prepared to administer.** AIX systems tend to be smaller than typical mainframe operating systems and may require a different viewpoint for administration and security. An AIX system is unlikely to have a formal security officer and probably does not have a full-time administrator of any kind.

AIX/6000 is designed to meet C2 security and, this document describes the administrative efforts necessary to install and maintain this class system.

The "Orange Book" specifications skip two especially important areas:

- Networking
- System updates

Most AIX systems are attached to local area networks, and it is sometimes difficult to clearly separate network security from individual system security. An individual system administrator has little control over general network security and must accept some network functions ″on faith.″ It is foolish, for example, to demand a B1 system and then connect it (with default, standard facilities) to an ″open″ TCP/IP network.[73]

Most commercially available operating systems have updates.[74] The DoD specifications seem to ignore these. It is not practical to re-certify a system for

[73] DoD defines a secure networking protocol named DoDIIS Network Security for Information eXchange (DNSIX), which restricts import and export of classified data through network interfaces. Early versions of AIX 3 provided network interface controls matching parts of these functions. These functions were rarely used in "real world" systems, and, in particular, were not used by generally-available TCP/IP functions. The TCP/IP community (as represented by the IETF (Internet Engineering Task Force))) has chosen other directions for future TCP/IP security mechanisms.

[74] IBM sometimes calls these ″PTF tapes″ or ″system maintenance tapes″ or ″system upgrades″ and so forth.

every update or ″fix″. Thus, in principle, a system is uncertified after any update/fix is installed. The same concept applies when third party software products having ″authorized″ modules are installed. In practice these upgrades and products are accepted on faith.

Several members of the European community have produced their own information processing security criteria definition, named ITSEC. This started with the basic elements from the Orange Book, and then went in somewhat different directions. A discussion of ITSEC can be found in *The Library for Security Solutions: Security Reference* (IBM publication number GG24-4106).

Appendix B. Additional Authentication

AIX allows you to specify additional primary authentication steps (“methods”) and secondary authentication steps. In AIX terminology, a primary authentication method can reject a user login; a secondary authentication method cannot reject a login. A secondary authentication step is a method for running a specific program (which may have nothing to do with authentication) as part of a specific user′s login process. (This terminology is unique to AIX.)

B.1 Two-person Login

One common method of increasing login security is to require two passwords. The assumption is that two different people (with the two different passwords) must be present to complete the login. The two different passwords are associated with two different accounts. There is no way, using the standard facilities, to maintain two passwords with a single account.

You can specify a two-person login for a specific account by setting the following parameters, using **smit**:

```
smit
  Security and Users
    Users
      Change/Show Characteristics of a User

      *User NAME                    [joe]
       ...
       PRIMARY Authentication Method   [SYSTEM,SYSTEM;mary]
```

When *joe* logs into the system, in this example, he will be prompted for his password. If he responds correctly, the system will issue a prompt for mary′s password. (Of course, *joe* might know both passwords, but this defeats the purpose of the two-person login.) The prompts would be:

```
login: joe
joe′s password: xxxxxxx
mary′s password: xxxxxxx
```

You must set the PRIMARY Authentication Method exactly as shown above. The *SYSTEM* parameter specifies that the normal password authentication program should be used. By default, it checks the password of the user currently logging into the system. The second *SYSTEM* parameter specifies a second check. In this case it has an operand, *;mary*, and verifies the password for the account specified in the operand. (The syntax is unusual: a comma separates two different authentication steps. A semicolon separates an authentication method (SYSTEM) from an optional account name. See the comment below before attempting to remove the SYSTEM method.)

B.2 Password and Local Program

You can add a local program that provides additional authentication. Your program obtains control during a login process. If your program ends with return code zero, the login process continues. If your program ends with any other return code, the login process is terminated with the message “You

© Copyright IBM Corp. 1994

entered an invalid login name or password.” The additional authentication program runs as *root*; it need not have *suid*.

A very simple program is shown here. You would, of course, have more complex programs, and you would probably not echo the user′s response to the screen (as the following program does).

```
#include <stdio.h>
char ans[256]
main()
{ printf("What is your cat′s name? ");
  fflush(stdout);
  fscanf(stdin,"%s",ans);
  if (strcmp(ans, "sylvester") == 0 ) exit (0);
  if (strcmp(ans, "archie"   ) == 0 ) exit (0);
  exit (-1);
}
```

This program (after you compile it, and place the output in a convenient location -- we used ***/usr/bin/cats***) must be defined as an *authentication method*. This is done by adding two lines to ***/etc/security/login.cfg***. Find the following lines in this file:

```
* auth_method:
*       program =
```

You can overwrite these lines (using your favorite editor) or add lines after these. Add the following stanza:

```
CATS:
    program = /usr/bin/cats      (or whatever name you used)
```

The authentication method name (“CATS”) need not be the same as the program name (“cats” in this example).

Next, use **smit** to add this authentication method to one or more users. (You can add it to the *defaults* stanza of ***/etc/security/users***, but it should be well tested before doing this.)

```
smit
  Security and Users
    Users
      Change/Show Characteristics of a user
        *User NAME                      [bill]

         PRIMARY authentication method   [SYSTEM,CATS]
```

When user *bill* next logs into the system, he should receive the prompts:

```
Console login: bill
bill′s password: xxxxxxx
What is your cat′s name?
```

If *bill* responds with “sylvester” or “archie,” the login will succeed. Any other response will cause the login to be rejected.

If you specify only CATS as the authentication method (that is, remove the SYSTEM parameter), the login process will first prompt for the cat′s name. It will then prompt for the standard password. That is, the SYSTEM authentication (using the standard password) apparently is always used, whether or not it is specified.

Do not delete the *auth1* parameter from the default stanza in ***/etc/security/user***. If you do, you will be unable to log into any account that does not have an additional authorization method defined. You can change the *auth1=SYSTEM* parameter in the default stanza to another method, such as *auth1=CATS*. In this case, the system will prompt every user (who does not have a specific auth1= parameter defined) for a cat's name (or whatever your authorization method does), and it will then prompt for his standard password. Apparently, you cannot bypass the standard password prompt by removing the *STANDARD* parameter in either the default or user's stanza.

If you experiment much with various authentication methods and parameters, you will probably lock yourself out of your system. If this happens, review 3.9.1, "Repairing the root Userid" on page 36. A good suggestion is to test the *root* recovery process **before** you start working with authentication parameters.

Appendix C. Audit Events

The following is a list of all known audit events built into AIX 4.1. A description of the user (or system) operation that triggered the event is listed, followed by the audit event name, followed by a brief description (in parenthesis) of the audit event if this is not clear from the event's name.

Note that some events do not represent a global view. For example, if a user account is added to the system by directly editing ***/etc/passwd***, a *mkuser* audit event is not generated.

```
USER/SYSTEM   AUDIT EVENT       TAIL

fork          PROC_Create       child process ID
exit          PROC_Delete       process ID
exec          PROC_Execute      euid, egid, epriv, name
setuidx       PROC_RealUID      real UID
setuidx       PROC_AuditID      login UID
setgidx       PROC_RealGID      real GID
usrinfo       PROC_Environ      environment buffer
sigaction     PROC_SetSignal
setrlimit     PROC_Limits
nice          PROC_SetPri       new priority
setpri        PROC_SetPri       new pripríty
setpriv       PROC_Privilege    command, privelege set
settimer      PROC_Settimer
open          FILE_Open         mode, descriptor, filename
create        FILE_Open         mode, descriptor, filename
read          FILE_Read         descriptor
write         FILE_Write        descriptor
close         FILE_Close        descriptor
link          FILE_Link         linkname, filename
unlink        FILE_Unlink       filename
rename        FILE_Rename       frompath, topath
chown         FILE_Owner        UID, GID, filename
chmod         FILE_Mode         mode, filename
mount         FS_Mount          object, stub
umount        FS_Umount         object, stub
chacl         FILE_Acl          acl
chpriv        FILE_Privilege    privelege
chdir         FS_Chdir          directory path
chroot        FS_Chroot         directory path
rmdir         FS_Rmdir          directory name/path
mkdir         FS_Mkdir          directory name/path
msgget        MSG_Create        key, msqid
msgrcv        MSG_Read          msqid, mpid
msgsnd        MSG_Write         msqid
msgctl        MSG_Delete        msqid
msgctl        MSG_Owner         msqid, UID, GID
msgctl        MSG_Mode          msqid, mode
semget        SEM_Create        key, semid
semop         SEM_Op            semid
semctl        SEM_Delete        semid
semctl        SEM_Owner         semid, UID, GID
semctl        SEM_Mode          semid, mode
shmget        SHM_Create        key, shmid
shmat         SHM_Open          shmid
```

© Copyright IBM Corp. 1994

```
shmctl       SHM_Close         shmid
shmctl       SHM_Owner         shmid, UID, GID
shmctl       SHM_Mode          shmid, mode
             TCPIP_config      (5 text strings)
{various     TCPIP_host_id     (4 text strings)
 modules     TCPIP_route       (5 text strings)
 in the      TCPIP_connect     (5 text strings)
 TCPIP       TCPIP_data_out    (5 text strings)
 subsystem}  TCPIP_data_in     (5 text strings)
             TCPIP_access      (5 text strings)
             TCPIP_set_time    (4 text strings)
             TCPIP_kconfig
             TCPIP_kroute
             TCPIP_kconnect
             TCPIP_kdata_out
             TCPIP_kdata_in
             TCPIP_kcreate
tsm          USER_Login        login command
tsm          PORT_Locked       port name
sysck        SYSCK_Check       (text string)
sysck        SYSCK_Update      (text string)
sysck        SYSCK_Install     (text string)
usrck        USER_Check        (3 text strings)
logout       USER_Logout       (text string)
chsec        PORT_Change       portname, valuea
chuser       USER_Change       (2 text strings)
rmuser       USER_Remove       (text string)
mkuser       USER_Create       (text string)
setgroups    USER_SetGroups    (2 text strings)
setsenv      USER_SetEnv       (2 text strings)
su           USER_SU           (text string)
grpck        GROUP_User        userid, group name
grpck        GROUP_Adms        userid, group name
chgroup      GROUP_Change      (2 text strings)
mkgroup      GROUP_Create      (text string)
rmgroup      GROUP_Remove      (text string)
passwd       PASSWORD_Change   (text string)
pwdadm       PASSWORD_Flags    (text string)
pwdck        PASSWORD_Check    userid, error, status
pwdck        PASSWORD_Ckerr    user/file name, error, status
startsrc     SRC_Start         (text string)
stopsrc      SRC_Stop          (text string)
addssys      SRC_Addssys       (text string)
chssys       SRC_Chssys        (text string)
addserver    SRC_Addserver     (text string)
chserver     SRC_Chserver      (text string)
rmssys       SRC_Delssys       (text string)
rmserver     SRC_Delserver     (text string)
enq          ENQUE_admin       queue, device, request, operation
qdaemon      ENQUE_exec        queue, request, host, file, operation
sendmail     SENDMAIL_Config   (text string)
sendmail     SENDMAIL_Tofile   from userid, to filename
at           AT_JobAdd         filename, userid, time
at           AT_JobRemove      filename, userid
cron         CRON_JobRemove    filename, userid, time
cron         CRON_JobAdd       filename, userid, time
cron         CRON_Start        event name, command, time
cron         CRON_Finish       userid, pid, time
nvload       NVRAM_Config      (text string)
```

```
cfgmgr        DEV_Configure    device name
chdev         DEV_Change       (text string)
mkdev         DEV_Change       (text string)
mkdev         DEV_Create       mode, device, filename
mkdev         DEV_Start        (text file)
installp      INSTALLP_Inst    option name, level, installation
installp      INSTALLP_Exec    option name, level, program name
updatep       UPDATEP_Name     (text string)
rmdev         DEV_Stop         device name
rmdev         DEV_UnConfigure  device name
rmdev         DEV_Remove       device name
lchangelv     LVM_ChangeLV     (text string)
lextendlv     LVM_ChangeLV     (text string)
lreducelv     LVM_ChangeLV     (text string)
lchangepv     LVM_ChangeVG     (text string)
lextendpv     LVM_ChangeVG     (text string)
lreducepv     LVM_ChangeVG     (text string)
lcreatelv     LVM_CreateLV     (text string)
lcreatevg     LVM_CreateVG     (text string)
ldeletepv     LVM_DeleteVG     (text string)
rmlv          LVM_DeleteLV     (text string)
lvaryoffvg    LVM_VaryoffVG    (text string)
lvaryonvg     LVM_VaryonVG     (text string)
backup        BACKUP_Export    (text string)
backup        BACKUP_Priv      (text string)
restore       RESTORE_Import   (text string)
shell         USER_Shell       (text string)

   ------------- Object Events -----------------

              S_ENVIRON_WRITE  /etc/security/environ
              S_GROUP_WRITE    /etc/group
              S_LIMITS_WRITE   /etc/security/limits
              S_LOGIN_WRITE    /etc/security.login.cfg
              S_PASSWD_READ    /etc/security/passwd
              S_PASSWD_WRITE   /etc/security/passwd
              S_USER_WRITE     /etc/security/user
              AUD_CONFIG_WR    /etc/security/audit/config
```

Index

Special Characters

/audit directory 85
/audit/auditb 85
/audit/stream.out 84
/audit/trail 56, 84
/dev modes 78
/dev ownership 78
/dev/console 59
/etc/exports 68
/etc/group 30
/etc/hosts.equiv 64
/etc/hosts.lpd 65
/etc/inetd.conf 66
/etc/passwd 13, 30
/etc/passwd usage 20
/etc/passwd.dir 30
/etc/passwd.pag 30
/etc/profile 24, 30, 32
/etc/security directory 20
/etc/security/.ids 16, 30
/etc/security/.profile 16, 30
/etc/security/audit files 85
/etc/security/audit/bincmds 84
/etc/security/audit/streamcmds 84
/etc/security/config 63, 66
/etc/security/environ 16, 30
/etc/security/failedlogin 30, 56
/etc/security/group 27, 30
/etc/security/lastlog 30
/etc/security/limits 30, 32
/etc/security/login.cfg 30, 35
/etc/security/mkuser.default 27, 30
/etc/security/oenviron 33
/etc/security/olastlog 33
/etc/security/olimits 33
/etc/security/opasswd 33
/etc/security/osysck.cfg 33
/etc/security/ouser 33
/etc/security/passwd 16, 30
/etc/security/sysck.cfg 78, 79
/etc/security/user 20, 30, 33, 95, 111
/etc/security.lastlog 76
/etc/utmp 56, 76
/set/hosts 66
/tmp 59
/tmp files, security 58
/var/adm/acct/fiscal/* 56
/var/adm/acct/nite/* 56
/var/adm/acct/sum/* 56
/var/adm/cron/at.allow 93
/var/adm/cron/at.deny 93
/var/adm/cron/cron.allow 93
/var/adm/cron/cron.deny 93
/var/adm/cron/log 56
/var/adm/dtmp 56
/var/adm/messages 56
/var/adm/pacct 56
/var/adm/qacct 56
/var/adm/sulog 15, 56, 76
/var/adm/wtmp 56, 75
/var/mail/* 56
.netrc 65, 103
.profile 24
.rhosts 65
$HOME permissions 100

A

access control 49
ACCOUNT LOCKED control 18
accounting 75
ACL editor 56
ACL, NFS 69
acledit command 53
aclget command 53
aclput command 53
ACLs, definition 53
ACLs, with groups 27
adapter security 73
administrative parameters 18
administrator, system 6
archive files 97
at, controls 93
atime, timestamp 52
audit classes 85
audit command 85
audit events 83, 115
audit objects 83, 115
audit subsystem 83
audit tail 84
audit, security 7
authentication 9
AUTHENTICATION GRAMMAR 19
authentication methods 111
authentication, additional 33
authentication, setting 19
authoriztion 9

B

B1 systems 105
B2 systems 105
base ACL 53
BIN mode, audit 84
boot, hard disk 4
boot, tape/CD-ROM 4

© Copyright IBM Corp. 1994

C

C1 systems 105
C2 systems 105
C2-level security 9
CD-ROM file system 40
chmod command 49, 56
chown command 58
communication lines 61
core 56
cp command 44
cracker programs 37
cron controls 96
cron, controls 93
cronadm command 94
crontab command 93
ctime, timestamp 52

D

data import security 73
data scopes 61
defaults, system 20
deny 53
DES, Secure NFS 70
DFS, auditing 89
dial-up security 61
directory permissions 4, 45, 100
DoD security levels 3

E

EDITOR variable 55
encryption, NFS 70
error logging 57
errpt command 57
Ethernet security 61
event auditing 83
EXPIRATION date 18
expiration, password 19
extended ACL 53, 56

F

file security, introduction 39
file system, private 41
file systems, definition 39
files, growing 56
files, unowned 59
find command 59
firewall 91
firewalls 62
fsck command 40
fsize 32
ftp command 63, 103

G

GID, file owner 43
group inheritance bit 51
group password 27
groups 27
groups, multiple 54
groups, standard 29
grpck command 33, 99

H

Haystack Labs., Inc. 90
herald, login 96
hubs, switched 61

I

IFS 36
IFS variable 97
INed files 57
initial PROGRAM 18
inodes 42
intruder, definition 4

J

JFS 39
journalled file system 39

K

kernel extensions 9
keyswitch, S/6000 4

L

LAN adapter security 73
LAN anaylzers 61
LAN channels 61
last command 76
lastlog file 76
LFS 40
line security 61
link permission bit 51
links, file 42
local file system 40
LOCKED control 18
LOGIN control 18
login herald 96
LOGIN times 18
login, authentication 111
login, two-person 111
logs, AIX 75
ls command, intro 47
lsgroup command 33
lsuser command 33

M

mkpasswd command 35
mkuser command 16
mode bits 44
mount command 39, 41
mount point 40
mtime, timestamp 52
mv command 44

N

netstat command 67
network file system 67
network information service 71
network security 61
network security goals 62
new user 97
NFS 67
NFS files 59
NFS, auditing 89
NFS, secure 69
NIS 71

O

object auditing 83
Orange Book 105
owner permissions 45
ownership, file 43

P

passwd command 21
password entry 21
password quality controls 95
password warning 19
password, cracker 37
password, encryption 21
password, guidelines 22
password, new user 97
password, quality 23
passwords, overview 21
passwords, problem 4
passwords, setting 21
passwords, two-person 111
PATH 24
permission bits 44
permissions, directory 45, 51
permissions, exclusions 50
permissions, file 45
permit 53
physical security 4, 61
port, control 35
port, SAK 80
power-on hours 6
profile 16
program access control 49
prompt, shell 26, 96
pwdck command 33, 99

Q

quotas, setting 19

R

RACF, MVS 9
rcp command 63, 103
REGISTRY 19
remote login 64
remote LOGIN control 18
remsh 103
reset button 4
rexec command 63
rlogin command 63, 103
root password 95
root user, definition 14
root user, sharing 14
root, disabling 26
root, repairing 36
rsh command 63, 103

S

SAK 79
SAK, all terminals 96
save-text bit 49
search permission 47, 52
secure attention key 79
securetcpip command 63, 96, 103
security audit 7
Security Policy 1
security structure 9
security, adapter 73
security, group 29
server, definition 3
SGID 53
sgid bit, definition 49
shadow files 20
shar files 97
shell prompt 26
shell script suid 50
skulker command 56
skulker script 92
skulker, output 96
smit, audit parameter 85
smit, description 11
smit, user control 16
smit, user menu 17
smit.log 56
sniffers, snoopers, LAN 61
SOCKS protocol 91
specify 53
Stalker product 90
sticky bit 47, 51, 59

sticky bit, definition 49
sticky bit, use 50
stopsrc command 63
STREAM mode, audit 84
su command 97
SU control 18
subsystems 9
SUID 53
suid bit, definition 49
suid, problem 4
suid, shell scripts 4, 50, 58
sulog 15
sulog file 76
superuser 29
SVTX 53
SVTX bit 51
switched hubs 61
symbolic link 43
symbolic links, permissions 58
sysck program 78, 79
sysck.cfg file 78, 79
system administrator 6
system structure 9

T

TCB 9, 77
tcb, recommendation 95
tcbck program 78, 79, 96
TCP/IP 61, 66, 103
telnet command 63, 103
tftp command 63, 103
threads, audit 89
TIMEOUT variable 25, 95
timeout, control 25
times, login, control 18
timestamps, file 52
TMOUT variable 25, 95
token ring security 61
tpath parameter 80
Trojan horse 25, 80
trpt command 63
trusted computing base 9, 77
trusted shell 79
tsh shell 79
tty, control, user 19

U

UID, creating 17
UID, definition 13
UID, file owner 43
ulimit parameter 58
umask 52
umask value 98
UMASK, setting 19
umount command 39
unowned files 59
user accounts 13
userid, creating 17
userid, definition 13
userids, standard 29
users 16
usrck command 33, 99
utmp file 76

V

VFS 40
virtual file system 40
visual system manager 11
vsm, description 11

W

warning, password 19
who command 36, 76
workstation security 15
workstation, definition 3
wtmp file 75

X

X Windows 92
XDM 92
XStation 120 63