X/Open Guide

Security Guide
(Second Edition)

X/Open Company, Ltd.

X/Open Guide

Security Guide (Second Edition)

X/Open Document Number: XO/GUIDE/90/010

Set in Palatino, Helvetica and Courier by X/Open Company Ltd., U.K.
Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

SECURITY GUIDE, SECOND EDITION

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring greater value to users through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable system environment, called the *Common Applications Environment (CAE)*. This environment covers all the standards, above the hardware level, that are needed to support open systems. It ensures portability and connectivity of applications, and allows users to move between systems without retraining.

The interfaces identified as components of the Common Applications Environment are defined in the *X/Open Portability Guide*. This guide contains an evolving portfolio of practical applications programming interface standards (APIs), which significantly enhance portability of application programs at the source code level. The interfaces defined in the X/Open Portability Guide are supported by an extensive set of conformance tests and a distinct trademark - the X/Open brand - that is carried only on products that comply with the X/Open definitions.

X/Open is thus primarily concerned with standards selection and adoption. The policy is to use formal approved *de jure* standards, where they exist, and to adopt widely supported *de facto* standards in other cases.

Where formal standards do not exist, it is X/Open policy to work closely with standards development organizations to encourage the creation of formal standards covering the needed functionalities, and to make its own work freely available to such organizations. Additionally, X/Open has a commitment to align its definitions with formal approved standards.

The X/Open Product Family - XPG

There is a single family of X/Open products, which has the generic name ''XPG''.

XPG Versions

There are different numbered versions of XPG within the XPG family (XPG1, XPG2, XPG3). Each XPG version is an integrated set of elements supporting the development, procurement and implementation of open systems products, and each comprises its own:

- XPG Specifications
- XPG Verification Suite
- XPG descriptive guides

- XPG trademark licensing materials

The XPG trademark (or ''brand'') licensed by X/Open always contains a particular XPG version number (e.g., ''XPG3'') and, when associated with a vendor's system, communicates clearly and unambiguously to a procurer that the software bearing the trademark correctly implements the corresponding XPG specifications. Users specifying particular XPG versions in their procurements are therefore certain as to the XPG specifications to which vendors' systems conform.

XPG Specifications

There are four types of XPG specification:

- **XPG*n* Formal Specifications**

  These are the long-life XPG specifications that form the basis for conformant/branded X/Open systems, and are the only type of XPG specification released with an XPG version number (e.g., ''XPG3''). They are intended to be used widely within the industry for product development and procurement purposes. Currently, all XPG Formal Specifications are included in Issue 3 of the X/Open Portability Guide.

  Individual XPG specifications are released as Formal Specifications only as part of the formal release of the complete XPG version to which they belong. However, prior to the launch of that XPG version, they may be made available as:

- **XPG Developers' Specifications**

  These are specifically designed to allow developers to create X/Open-compliant products and applications in advance of the formal launch of a future version of the XPG.

  Developers' Specifications may be relied on by product developers as the final, base specification that will appear in a future XPG. They are made available beforehand in order to meet the need of product developers for advance notification of the contents of XPG Formal Specifications, to assist in their product planning and development activities.

  By providing such advance notification, X/Open makes it possible for products conforming to future XPG Formal Specifications to be developed as soon as practicable, enhancing the value of XPG itself as a procurement aid to users.

- **XPG Preliminary Specifications**

  These are XPG specifications, usually addressing an emerging area of technology, and consequently not yet supported by a base of conformant product implementations, that are released in a controlled manner for validation purposes. A Preliminary Specification is not a ''draft'' specification. Indeed, it is as stable as X/Open can make it, and on publication will have gone through the same rigorous X/Open development and review procedures as XPG Formal and Developers' Specifications.

  Preliminary Specifications are analogous with the ''trial-use'' standards issued by formal standards organizations, and product development teams are intended to develop product on the basis of them. Because of the nature of the technology they are addressing, they are untried in practice, and they may therefore change before being published as an XPG Formal or Developers' Specification.

- **Snapshot Specifications**

  These are ‘‘draft’’ documents, that provide a mechanism for X/Open to disseminate information on its current direction and thinking to a limited audience, in advance of formal publication, with a view to soliciting feedback and comment.

  A snapshot represents the interim results of an X/Open technical activity. While X/Open currently intends to progress this activity towards publication of an X/Open Guide, X/Open is a consensus organisation, and makes no commitment regarding publication.

  Similarly, a snapshot does not represent any commitment on behalf of any X/Open member to make any specific products available now or in the future.

Guides

X/Open Guides provide information that X/Open believes will be useful in the evaluation, procurement, development and/or management of open systems, particularly those that are X/Open-compliant.

X/Open Guides are non-normative, and should not be referenced for purposes of specifying or claiming X/Open-conformance.

This Document

This document is an X/Open Guide (see above). It provides an introduction to security and related topics for systems that conform to the X/Open Common Applications Environment. It is intended for all users of the computer and supplements installation and management documents provided by the vendor of the system.

The Guide contains background material on security, safeguards, programming practices, preventive measures, auditing and other topics related to security. Some topics relate directly to X/Open-compliant systems. Some also apply to most other general-purpose multi-user operating systems. This Guide assumes that the reader has a working knowledge of the concepts of the X/Open operating system interfaces, as described in the **X/Open Portability Guide**. However, **Chapter 2**, **Security Mechanisms** re-introduces the concepts that apply to security.

This is the second edition of the X/Open Security Guide. The major change for this edition of the Guide is to align with the **X/Open Portability Guide**, **Issue 3**. Many of the examples are changed to use only interfaces specified by the **X/Open Portability Guide**, **Issue 3**.

Although this guide is based on the **X/Open Portability Guide**, **Issue 3**, the majority of the issues discussed are expected to relate equally to subsequent editions. Where it is necessary to reference features beyond those specified in the **X/Open Portability Guide**. for example systems administration facilities, references have been illustrated with examples based on UNIX System V.

Trademarks

X/Open™ is a trademark of the X/Open Company Limited.

UNIX® is a registered trademark of USL Inc. in the USA and other countries.

Referenced Documents

The following documents are referenced in this guide:

- UNIX System V - Release 2.0 Programming Guide (April 1984 - Issue 2)
- Trusted Computing Security Evaluation Criteria, DoD 5200-28-STD (revised in December 1985)

Chapter 1

Introduction to Security

1.1 COMPONENTS OF SECURITY

1.1.1 Availability

A computer system represents value that must be protected. The system's value includes the value of the hardware and accessories, the value of the time that users spend working on the system, and the value of the time and money that the organisation intends spending on the system, assuming that it will continue to operate correctly. Protection of this value requires continuing *availability* (or continuity of service) of the system. Loss of this availability is called ''denial of service''.

1.1.2 Integrity

The system's value also includes the accumulated value of the software at the installation, and the value of the data that resides in the system. The value of these objects (data files and programs) lies in their *integrity* (i.e., in the preservation of their contents against all unauthorised change). An object that has lost its integrity is said to be *corrupted*. The change to the object may have been accidental or it may have been deliberate and unauthorised. The corruption of certain objects, such as the user database, has a special significance, since it raises the possibility of corruption of additional objects and additional loss of value.

The concept of integrity also applies to the computer as a whole. For a system to retain its integrity, it must preserve the value of its components, i.e.:

- It must remain able to perform its ''file storage'' functions.[1] (A ''corrupted file system'' is a file system whose internal accounting of data objects, blocks and attributes has somehow failed, which raises the possibility of valuable objects being lost in the future.)
- It must remain able to perform its security functions: the features that protect the system's value must continue to work correctly.[2]

1.1.3 Confidentiality

For certain objects, integrity alone does not ensure the object's value. For example, the value of files containing marketing data would diminish if their contents were known to a competitor. For the object to retain its complete value, it must retain not only integrity but also confidentiality, i.e., not be accessible to unauthorised persons.

---

1. This technical definition of integrity is discussed further in **Section 6.5.2**, **File Systems**.
2. This is the primary meaning of integrity for the purposes of this Guide.

1.1.4 Security

Security comprises availability, integrity and confidentiality, and everything about the computer and the organisation's use of it that protects its value. It requires the cooperation of the administrator, programmers and users. Certain X/Open software features relevant to security are discussed in this Guide, as are the correct use of these features and other procedures that the organisation can take (such as physical security) to ensure that the software security features cannot be defeated.

Security policy comprises the organisation's rules that protect the value of the computer. If the organisation has addressed security in a deliberate way, the security policy typically takes the form of a written document. If not, the actual policy may be a combination of individual initiative and tradition, and the effectiveness of the policy may be questionable.

1.1.5 Threats to Security

A *penetration* is an act that bypasses or disables security on a computer. The *penetrator* is the person who commits or initiates the act.

The penetrator may be a member of the organisation or an outsider. The penetrator's goal may be denial of service to other users (sabotage or vandalism of the system), access to unauthorised data, personal profit (through computer fraud or unauthorised use of computing services) or to save time by bypassing the organisation's security discipline. Alternatively, there may be no goal at all or the penetration may be inadvertent.

Although analysing a penetrator's motives may help the administrator solve a penetration or guard against specific penetrations (notably taking precautions when an employee leaves the organisation), this analysis does not change the general security policy. It is useful to assume, and plan as though, every penetration is malicious and designed to disable the entire system.

The administrator seeks to prevent successful penetration of the computer mainly by enforcing correct use of the X/Open security features. Penetration attempts can also be dissuaded by having an effective policy that includes auditing (see below). The administrator may also protect the system by doing certain things expressly to confuse or bluff a penetrator, and usually cooperates in efforts to discipline a penetrator, in order to maintain the effectiveness of the security policy.

1.1.6 Types of Penetration

A simple form of penetration occurs when one person assumes the identity of another so as to perform an unauthorised action. Typical X/Open-compliant systems inhibit this by the use of passwords.

Automated penetration is more troublesome. It involves placing on the system a program that threatens security. By using a program, the penetrator may achieve denial of service or may corrupt objects at a later time or under a different identity. This makes it difficult for the administrator to associate the damage with the penetration.

Overt automated penetrations, programs that immediately corrupt objects, can usually be traced in a straightforward way. Automated penetrations that are harder to trace are the following:

- A *Trojan horse*. This is a program that apparently performs a useful task but carries out other actions covertly, e.g. using the authorisations or rights of the caller to circumvent the security policy. A special case of a Trojan horse is a *spoofing program*, which tricks the user into typing a password, which it then makes available to the penetrator.
- A *virus*. This is an addition to a program with replicating capability. A typical virus has the ability to damage objects, as well as the ability to hide a partial or full copy of itself in one or more unrelated programs. Its replication amplifies the damage it does. The copies may be executed at different times, by different users and often with greater privileges than at the time of original penetration. Many viruses are written so as to have a delayed effect, thereby further hampering the administrator's efforts to associate the damage and data loss with the penetration event. After penetration by a virus, restored assurance of a system's integrity may require analysis or recompilation of all programs changed since the penetration, since they could contain copies of the virus.
- A *worm*. One or more malicious programs that replicate themselves throughout a computer system or network. A worm is similar to a virus but does not need to reside in a host program.

1.1.7 Requirements of Effective Security

Effective security requires user education, documentation, assurances, management support, auditing and effective use of suitable software features.

- *User education* is necessary to ensure that an installation security policy is followed throughout all aspects of system use. It may be necessary to ensure secure access and use of buildings, rooms, and facilities such as terminals. User awareness may also be necessary for the effective use of software facilities, such as periodic password change, resource sharing, and access control mechanisms.
- *Documentation* communicates the policy to all users, since the policy requires their cooperation.
- *Assurances* are standards or procedures by which the organisation can verify that the policy has been implemented correctly and is achieving security. In the **X/Open Portability Guide**, typical assurances are that appropriate privileges are required to access privileged system interfaces (such as *setuid*( )), and that these privileges cannot easily be used by normal users.[3]
- *Management support* is required to define the security policy and to get users to cooperate in implementing it. Management must establish that protecting the value of the computer's assets is part of every user's job.
- *Auditing* means recording events that are relevant to security. These events may include evidence of security breaches. Auditing indirectly increases a system's

3. This gives rise to the concept of an administrative super-user. This term is used frequently in the remainder of the Guide when referring to a user with appropriate authorisations.

security by letting the administrator detect and reconstruct security failures. The administrator can then react to prevent recurrence. The existence of auditing in the security policy increases the accountability of users and dissuades penetration attempts. However, unrestrained use of audit capability can potentially affect system performance and availability.

- *Software security features* contribute to the enforcement of a system's security policy by requiring authentication, access control, isolation, accountability and privilege.
  - — *Authentication* means ensuring that users are who they claim to be, typically by the use of passwords.
  - — *Access control* means ensuring that authenticated users can gain access to the resources they need, and that they cannot gain access to other resources.
  - — *Isolation* means separating users from each other, so as to keep users from gaining access to other users' resources. By denying a user or process access to such data, it prevents inadvertent corruption of objects and hinders their malicious use.
  - — *Accountability* means ensuring that the use of security-related features can be traced to the person that used them.
  - — A *privilege* is the power to perform some action on the computer that is not generally available. The ''least privilege'' principle states that processes that need to take a certain action: (1) receive the lowest privilege level that lets them do so, and (2) receive it only for the time they need it.

Chapter 2, **Security Mechanisms** describes how X/Open-compliant systems provide these software security features.

The goal of a security policy is to be able to assure users and management that the computer does and can continue to protect the availability, integrity and confidentiality of its components.

1.2 TYPES OF USER

Informally, a user is anyone who uses the computer. **Section 2.1**, **Users** defines ''user'' formally. Generally, each person has one login name in the user database. It does not compromise security for a person to have more than one login name.[4] But it defeats security if several people share a single login name, since this undermines personal accountability for acts that harm security.

This guide divides security tasks and responsibilities according to the type of job a user is doing on the computer. In particular, security places different requirements on administrators, system programmers and application users. **Chapter 3**, **Security for Users** discusses the security responsibilities of all users, **Chapter 4**, **Security for Programmers** is intended for system programmers, while much of the rest of the book covers administration tasks. Note that both system programmers and administrators are also users of the computer.

1.2.1 Administrator

This Guide, in referring to the administrator, assumes that there is a single individual with ultimate responsibility for the correct and secure operation of the computer.

The administrator's managerial significance is as the primary proponent of security, i.e., the person who approves, implements and enforces the security policy, and who is presumed to have authority to use policy to assure a secure system.

The administrator does many things to manage the computer. (**Chapter 6**, **Administrative Procedures** describes these actions.) The administrative manager may delegate some of these functions. The title ''administrator'' applies to the administrative manager or to any assignee performing an administrative function on the computer.

All administrative actions have an impact over one or more of the forms of security discussed in **Section 1.3**, **Forms of Security**. For example, installing software requires you to acquire the appropriate privilege; this affects software security. Backing up discs requires physical access to security-sensitive devices. Managing accounting and auditing records is also a job that greatly influences security.

Administrative Roles

There are actually several administrative roles involved in managing a time-sharing system. The security administrator is responsible for the security of the system. The system administrator manages user accounts, the operating system, the basic command set and the libraries. Other roles maintain individual spooling systems and application packages such as *mail* and *uucp*. Pseudo-users correspond to each of these specialised administrative roles. (**Section 6.1.3**, **Pseudo-Users** lists typical pseudo-users.) A user switches to the pseudo-user account when serving in that role. This policy helps implement least privilege. The security administrator must assign ultimate responsibility for use of a pseudo-user account to a specific person. Operators require

4. The concept of *group* (see **Section 2.2**, **Groups**) can be used instead to effect least privilege if a given individual needs to use the computer for different purposes at different times.

special authorisations but perform only specialised tasks, such as backing up disks and doing routine maintenance. The principle of least privilege dictates that a system separates the administrative role into as many of these specific roles as it can, that a user should assume only the role needed, and only for the time needed, to do a certain job.

1.2.2 System Programmers

System programmers develop programs and scripts that have security implications. Any widely used program acquires security implications because users invest a lot of time assuming the continuing availability and integrity of the program.

Although system programmers need no special security capabilities other than access to their source code, their products must be studied for security implications, as described in **Section 4.1**, **Programming Management**. Programmers and the administrator must agree that installation of the new program will not corrupt the system or open it to penetration.

1.2.3 Application Users

Application users are users who do not fit any of the preceding roles. Their role never requires special authorisations. Programmers may fit this role when their product does not become part of the operating system or command set.

1.3 FORMS OF SECURITY

There are seven forms of security that may be required. Defects in one can sometimes be offset by tightening safeguards in another. Although this Guide concentrates on software security, you should understand all forms. (See **Chapter 5**, **Managing Security** for additional information on determining the requirements for your system.)

1.3.1 Physical Security

Physical security protects the physical plant and equipment, using guards, alarms, cipher locks, concrete walls, etc.

Physical security is the only form of security that provides protection against physical damage to the system; no amount of software security can assure the availability and integrity of the computer and its data if the computer facility can be vandalised.

Although this Guide primarily discusses security against human penetration of the system, physical security measures should also protect the system from other types of physical damage, for example through fire or other natural disaster.

1.3.2 Emanation Security

Emanation security protects confidentiality by limiting electromagnetic emissions from computing devices and should be considered together with physical security. Without emanation security, sophisticated devices outside a facility can detect emissions and reproduce data streams or video screen images. Devices that display highly sensitive data should be identified and placed to the inside of buildings, away from areas like roads or parking lots that are difficult to observe.

Remember to consider visible emissions. The ability of unauthorised persons to view sensitive data (notably, viewing video screens through windows) also threatens confidentiality. The remedies are the same: control of physical placement or use of shielding.

1.3.3 Administrative Security

Administrative security means establishing and enforcing policies that determine who gets to do what with the system. It implements the principle of least privilege, and is a result of management's analysis of the reason for the system and of each user's responsibilities.

1.3.4 Personnel Security

Personnel security is related to administrative security. It describes the ways in which security policy applies to actual users.

The organisation typically applies a general security policy to its personnel. It may regulate entry to work areas and require identification badges. System security includes some of the same steps, implemented in concert with the organisation's other security procedures.

Personnel security requires two-way communication on the part of the administrator. For instance, policy may designate interaction between the administrator and the Personnel Department. In this way, the administrator learns of new users and of what

authorisations or roles they need. Similarly, the Personnel Department undertakes to notify the administrator when a computer user leaves the organisation. The administrator must educate all users, since security depends on their awareness of the rules and likelihood of obeying them.

1.3.5 Software Security

Software security includes features of the operating system and supporting programs that contribute to the implementation of the security policy through functions such as identification and authentication, resource access control, audit, integrity checks, etc.

1.3.6 Hardware and Firmware Security

This computing level executes the code of the operating system. The difference between hardware and firmware (between circuitry and a fixed microprogram) is not relevant to security; typically, the vendor assures the integrity of this computing level and the installation never changes it. We refer to this level as ''the machine''.

Since many programming environments give users access to the machine to let them write faster programs, the machine must implement sufficient isolation to keep users from bypassing software security. For example, ordinary processes must not be able to directly input, output or gain access to unrelated memory. Typically, the machine runs ordinary processes in a non-privileged ''user mode''. It keeps from users any capability that has security implications, except the ability to request service from the operating system.

To assure security, the machine must have enough fault-detection hardware to assure that it is maintaining isolation.

1.3.7 Network Security

Network security is in its infancy, but is of growing concern and will be addressed in greater detail by later issues of this Guide.

Most computers are linked to remote computers using Local Area Networks (LANs) and Wide Area Networks (WANs). Remote computers typically have different security policies and controls.

The administrator must investigate the level of security that each remote computer maintains. He can then decide whether or not to connect it, and, if it is to be connected, what confidence to place in it. A LAN within a single organisation usually maintains a unified security policy for all users of any machine, whereas a WAN in which there is a link to a machine in a foreign country, poses a threat from remote security breaches. Some installations treat all remote systems as foreign and hold received data until it has been checked by administrative personnel. This approach helps to assure security but is expensive to implement.

The administrator must distinguish data links he does or does not control. Depending on the anticipated threats he should decide whether to restrict users' ability to send sensitive information through those links.

Network security also depends on the degree to which the network is integrated into the operating system. Some systems offer syntax that makes remote machines appear as part of the local file system, others operate networks as an application, such as *uucp*.

1.4 HISTORY OF UNIX SECURITY

UNIX was developed at Bell Laboratories in the early 1970s, to be used by a small group of cooperating users. At that time, security was not a major concern. In the two decades since then, however, UNIX has been embraced by both the commercial and public sectors, has been adapted to machines ranging from personal computers to supercomputers, and now runs commercial, scientific and government computing applications.

In the 1980s, when systems based on UNIX and its derivatives began to proliferate, vendors began to sell administrative enhancements that allowed control over the use of the system.

The UNIX operating system and the Common Applications Environment have become a standard for government procurements. These users typically require the system's security to be precisely specified. In the United States, the Department of Defense (DoD) has required security enhancements on systems it procures. A 1983 DoD publication *Trusted Computing Security Evaluation Criteria*[5] is an important definition of security requirements for the industry. TCSEC classifies computer systems from Division D, the least secure, to Division A, the most secure. Division C is divided into two classes; Division B is divided into three classes. Classes within a division are identified by a number starting from 1; higher numbered classes specify more security features than lower numbered ones. Official ratings are issued by the US National Computer Security Center (NCSC), which evaluates submissions.

X/Open-compliant systems meet most of the requirements of Class C1 (Discretionary Security). Some vendors offer enhancements in line with the requirements of Class C2 (Controlled Access Protection) and beyond. These enhancements are outside the scope of this manual.

---

5. *Trusted Computing Security Evaluation Criteria*, DoD 5200-28-STD, revised in December 1985.

Chapter 2

Security Mechanisms

An X/Open-compliant system implements security by preventing certain things from happening. Every case where a process tries to perform an operation on an object may have security implications. The system decides whether to permit each, based on:

- Certain attributes of the process[1] (see **Section 2.3**, **Processes**)
- Certain attributes of the object (see **Section 2.4**, **Objects and Permissions**)
- The type of operation the process is attempting

These attributes normally depend on login information. They include descriptions of users (see **Section 2.1**, **Users**) and of groups (see **Section 2.2**, **Groups**). On this basis, an organisation can relate system security to its own policy on personnel, departments and job assignments.

1. Abstract discussion of security technique often uses the symmetrical terms ‘‘subject’’ and ‘‘object’’ in describing sensitive events. However, for our purposes, a subject will always be a process.

2.1 USERS

The **X/Open Portability Guide** describes a generic user database which is used to hold information about registered users of the system. This includes each user's login name, numerical user ID, numerical group ID, initial working directory, and the path name of the initial user program. The format and location of password information held in the authentication database is implementation-defined, although many systems combine the user and authentication data in a single system file **/etc/passwd**. This file is controlled by the system administrator and contains one line per user as follows:

```
sven:23NUaYaGaBrr6:113:100:Sven Svensson (Employee 1141):/users/sven:/bin/sh
```

There are seven fields in each line, separated by colons: (1) the user name that identifies a user to the computer; (2) an encrypted form of the password the user must type to prove identity;[2] (3) a number called the user identifier (UID), which should be unique for all users; (4) a second number, not necessarily unique to that user, called the group identifier (GID) (see **Section 2.2**, **Groups**); (5) an extra field which typically contains arbitrary personal identification; (6) the initial working directory; and (7) the path name of the program to be executed after a login, for example, the user's default shell.

2.1.1 Super-User

The **X/Open Portability Guide** is not specific about how appropriate privileges are determined. However, a common implementation is to define that UID 0 denotes the super-user, who automatically bypasses most security checks on the system. On such systems, there must be at least one entry in the user database with 0 specified as the UID. Typically, the user name for this user is ''root''. There may be additional entries in the user database that define other accounts with super-user privilege.

2.1.2 Uses

The user database is checked any time a user must be authenticated. Typically, a *login* utility authenticates a user at a previously unused terminal. (After successful authentication, the user is said to be ''logged in''.) On some systems, a */bin/su* utility lets a logged-in user create a shell that will operate as a different user. Both utilities generally require that the user specify a user name and a password. The user name typed selects an entry from the user database; the password typed, after encryption, must match the one in an associated authentication database. The other fields in the user database govern the response to a successful login. For example, the user program field lets the system respond differently to different users. These remaining fields help implement isolation, by providing information the system can use to separate users and data.

---

2. Although the contents of the password field in **/etc/passwd** are encrypted, the file is open to general access. A computer system classified in Division C (see **Section 1.4**, **History of UNIX Security**) must protect authentication data from unauthorised access. X/Open-compliant systems designed to qualify for Division C typically do not use the password field, but store passwords in a separate file. Other X/Open-compliant systems may also do so in future releases.

2.1.3 Changes

Though not guaranteed on all X/Open-compliant systems, users may be allowed to change their password using the *passwd* utility. The administrator has exclusive control over the other fields in the user database.

2.2 GROUPS

A ‘‘group’’ is a set of users. As described previously, the user database has a numerical group identity (GID) that the user acquires after successfully logging-in; this GID should be valid and meaningful. The placement of a GID in the user database asserts that the user is a member of the specified group.

2.2.1 Rules

Every user must be a member of at least one group and can be a member of several groups - the user’s login group, as well as others. (**Section 3.3.5**, **Groups** explains how a user switches between groups.) An organisation can decline to use the group feature by creating a different group for each user. Where several users are members of the same group, such a group typically represents a department or project in the organisation.

Authorised groups are defined in a group database. Again the **X/Open Portability Guide** is not specific about how this database will be supported, but typical implementations use the system file **/etc/group**. Each line in **/etc/group** defines one group, just as each line in **/etc/passwd** defines a single user. A typical line looks like this:

```
xopen::100:sven,bengt
```

The lines in this file contain four fields, again separated by colons: (1) the alphanumeric group name; (2) an encrypted form of the group password;[3] (3) the group identity (GID), also used in **/etc/passwd**; and (4) a list of names, separated by commas, of the users who are authorised to be members of this group.

Compare the preceding examples of lines from **/etc/passwd** and **/etc/group**. The line from **/etc/passwd** asserts that user ‘‘sven’’ (UID 113) is a member of group 100. The line from **/etc/group** cites ‘‘xopen’’ as the group name of group 100, and confirms that ‘‘sven’’ is one of its authorised members. Indeed, each GID used in **/etc/passwd** should refer to a group that is validly defined in **/etc/group**, and **/etc/group** should confirm that the user is an authorised member of that group. Utilities like *login* and */bin/su* may read both files to determine whether to grant a user access.

2.2.2 Other Uses

Utilities that list or manipulate group information, including *ls*, *newgrp* and *chgrp*, read the group database in order to translate between GIDs and group names.

2.2.3 Changes

The administrator has exclusive control over the contents of the group database.

---

3. This is for an unimplemented feature meant to be used with commands such as *newgrp*. The **X/Open Portability Guide**, **Issue 3**, **Volume 1**, **XSI Commands and Utilities** discourages the use of group passwords and notes (page 192) that ‘‘Group passwords may disappear in the future’’. The double colon in the example of a group file shows a null password for this group.

2.3 PROCESSES

Logging in changes the security characteristics of the process that controls the terminal. Executing a utility such as */bin/su* creates a new process with specified security characteristics. These characteristics are:

- Real UID and GID (the UID and GID obtained from the user database as a result of authenticating the user)
- Effective UID and GID (the UID and GID currently in effect)
- Optionally, a list of supplementary GIDs[4]
- Mask (restrictions on the attributes applied to objects created by the process - see **Section 2.4**, **Objects and Permissions**)

2.3.1 Changes

Utilities like *login* and */bin/su* change both the real and the effective UID and GID. The calls *setuid*( ) and *setgid*( ) change the effective UID and GID.[5] On some systems, users can change GID by invoking *newgrp*. An unprivileged process never changes its real UID and GID until a new user logs in.

The mask can be changed by the call *umask* and by the shell command *umask*.

2.3.2 Subprocesses

A process can create other processes using *fork*. These are known as subprocesses, or child processes, of the original process (which is known as the parent). Notably, the shell typically effects a user command by creating a process to take the appropriate action. When one process creates another, the subprocess inherits most of the parent's current attributes.

---

4. A process may have up to {NGROUPS_MAX} supplementary GIDs, where {NGROUPS_MAX} is a system constant defined in <**limits.h**>. These are used in determining file access permissions, in addition to the effective GID. If {NGROUPS_MAX} is set to zero, this feature is not implemented.
5. Another case where the effective UID and GID changes is when one of the *exec* functions is applied to an object with the set-user-ID and set-group-ID attributes. This is discussed in **Section 2.6**, **Set-User-ID Programs**.

2.4 OBJECTS AND PERMISSIONS

An object, such as a file, has associated attributes, several of which relate to security. They specify the object's owner, group and permissions. An object's permissions say who can do what with it, and are based on the nature of the desired action and on the identity of the process attempting the action. An object's owner and group let the object specify different permissions for different users and groups.

There are three types of action a process can attempt on the data in an object: reading it, writing it and executing it.[6] The *ls* command indicates read, write and execute permission using the letters ''`rwx`'', always in that order. The letters assert that the corresponding type of access is allowed. If a particular type of access is prohibited, the character ''-'' appears in place of the letter. For example, ''`r--`'' describes read-only access. The symbol ''`---`'' describes a total lack of access.

Many commands use an octal digit (from 0 to 7) to stand for a group of three permissions. There are eight possible states of the three flags:

Octal	Flags	Meaning
0	`---`	No access
1	`--x`	Execute permission
2	`-w-`	Write permission
3	`-wx`	Permission to write and execute (rare)
4	`r--`	Read permission
5	`r-x`	Permission to read and execute
6	`rw-`	Permission to read and write
7	`rwx`	Total access

An object's owner and group serve to divide the access rights of processes into three classes:

- *The owner* A process whose effective UID matches the object's UID.
- *The group* A process whose effective GID, or one of its supplementary GIDs, matches the object's GID.[7]
- *Others* All processes that do not fit either of the above cases.

An object's complete permissions are expressed as three concatenated groups, signifying the owner, the group and the others, respectively. The owner usually has ''`rw-`'' (6) or ''`rwx`'' (7) access. An object for which other processes have read-only access may have these permissions:

`rw-r--r--` (644)

---

6. For special cases of execute permission, see **Section 2.5**, **Access Rules**.
7. Ownership overrides group membership. An object's group permissions never apply to its owner, unlike its owner permissions.

An object that is usable only by its owner may have these permissions:

`rwx------` (700)

2.4.1 Initial States

When a process creates an object, the object receives a copy of the process's effective UID and GID. So the owner of a process owns any object the process creates. The process's mask (see **Section 2.3**, **Processes**) restricts the object's permissions.[8]

2.4.2 Changes

A process can call *chown*( ) and a user can execute *chown* and *chgrp* to change an object's UID or GID.

8. A process has options in *chmod*, *mknod*, *open* and *creat* that override these rules.

2.5 ACCESS RULES

2.5.1 Ability to Change Attributes

A process can change an object's permissions if the process's effective UID matches the object's UID (that is, if the process owns the object), or if the process has appropriate privileges.

A process can change an object's UID and GID if {POSIX_CHOWN_RESTRICTED}[9] is not set and the process's effective UID matches the object's UID, or if the process has appropriate privileges.[10] Otherwise, if {POSIX_CHOWN_RESTRICTED} is set, only processes with appropriate privileges are allowed to change the UID of an object, and the GID can only be changed, by a process without appropriate privileges, to either its effective GID or one of its supplementary GIDs.

2.5.2 Access to Processes

A process can send a *signal* to another process. As a special case, a process can kill another process. A process can send a signal to another process if the real or effective UID of the sender matches the real or effective UID of the receiver, or the sender has appropriate privileges.

2.5.3 Access to Objects

When a process tries to read, write or execute an object, the process's effective UID, effective GID, supplementary GIDs and the type of access desired is compared with the object's UID, GID and permission for the attempted type of access (read, write or execute). The decision has four stages:

1. If the process has appropriate privileges, access is permitted.
2. Otherwise, if the process owns the object (process UID = object UID), the object's owner permission permits or denies access.
3. Otherwise, if the process is in the same group as the object (process GID or one of the supplementary GIDs = object GID), the object's group permission permits or denies access.
4. Otherwise, the object's other permission permits or denies access.

2.5.4 Access to Devices

Access to devices is governed by objects called ''device special files''. They have attributes specifying owner, group and permissions, as files do, and the rules are the

---

9. For the definition of {POSIX_CHOWN_RESTRICTED} refer to the interface definition for *chown*() in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System**, **Interface and Headers**.

10. Changing the object's UID means that you no longer are the owner. Unless you have appropriate privileges, this operation is irreversible; i.e., after ceding ownership of an object, you can no longer execute the call necessary to reclaim ownership of the object.

same. For example, the ability of a process to use a magnetic tape drive depends on its ability to use the device special file that represents the tape drive.

2.5.5 Access to IPC Objects

Optional inter-process communication (IPC) objects include semaphores, shared memory segments and messages. These objects have a creator as well as an owner. Therefore, they have two sets of UID and GID. A process qualifies for owner access if its effective UID matches either the object's owner UID or creator UID. Otherwise, the process qualifies for group access if its effective GID or one of its supplementary group IDs matches either the object's owner GID or creator GID. Otherwise, the process qualifies for other access. For more details, see **Section 2.7**, **Inter-Process Communication** of the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers**.

2.5.6 Access to Directories

Directories, like other objects, have attributes indicating UID and GID. These are derived from the object's creator - typically, the process invoking *mkdir*. The permissions pertain only to the information in the directory itself: to the filenames and pointers to the object with which each filename is associated. The permissions do not apply to the objects themselves. An object's permissions are stored with the object; they are not in any directory. (You can think of an object as ''inside'' a directory; however, the directory does not contain any objects, it simply points to them.)

Read permission on a directory is the permission to display the names of the objects in the directory regardless of whether you have permission to read the objects themselves. Write permission on a directory is permission to change the list of files in a directory, for instance, to create and delete files.

Execute permission on a directory is called ''search permission'', and is permission to use the directory as part of a pathname and to *chdir*( ) into that directory.

The *rm* and *mv* utilities depend on the access rights to the directories, not on the access rights of the objects themselves. You can move and delete an object if the directory permits you to do so, even if you have no permission to the object itself.

It may seem that the directory's permission is superior to that of the object itself as you need search access to the directory simply to gain access to the object. However, the pointer in the directory need not be the only pointer to the object; the latter may have several valid pathnames and filenames. A user may be kept from using some of these but be allowed to use others.

2.6 SET-USER-ID PROGRAMS

Two flags in addition to those discussed in **Section 2.4**, **Objects and Permissions** are the set-user-ID and set-group-ID attributes. If a process uses one of the *exec* functions to execute an object that has one or both of these attributes, the process's effective UID or GID changes to the object's owner or group, respectively.

A program is typically given the set-user-ID attributes, so as to allow its users the same access as the program's owner has to certain objects, such as a database.

Section 3.3.3, **Set-User-ID Programs** discusses the danger to security that arises when a user casually applies the set-user-ID attribute to a program. **Section 4.4**, **Privileged Programs** describes precautions you should use when writing a set-user-ID program.

The effective UID and GID of a process retain their values unless changed. Although the system remembers the real (login) UID and GID, a process's effective UID or GID never implicitly lapses back to the real UID or GID.

A subprocess's UID or GID may differ from that of its parent if it executes a set-user-ID or set-group-ID program or explicitly calls *setuid*( ) or *setgid*( ).

2.7 PRIVILEGES

The execution of certain function calls and function call options requires special privileges. Only a process with appropriate privilege can perform such a function or function option. A process with special privileges is called a *privileged process*.

A program may have special privileges. Such a program is called a *privileged program*. The privileges of a *privileged program* may be inherited by a process executing the program. Examples of *privileged programs* could be programs that are set-user-ID with respect to certain UIDs.

A process may also inherit privileges from its parent process, and a process may change its own privileges, provided it has appropriate privilege to do so.

Certain programs (e.g., *login* and *su*) change the privileges of the executing process as the result of user authentication. A process executing as a user agent of a user with special authorisation will typically inherit special privileges even though it executes a non-privileged program.

Earlier parts of this chapter mentioned cases where security checks are waived for processes with appropriate privilege. This section further discusses the ability of processes with special privileges.

2.7.1 File Access

For processes with appropriate privileges, access checking procedures as described in **Section 2.5**, **Access Rules** do not apply. They can read, write or execute any object in the file system, provided that the object exists and that the attempted operation is reasonable (this is defined later in this chapter). A process with appropriate privilege can change the security attributes of an object even if it is owned by someone else.

2.7.2 Directories

For processes with appropriate privileges, access checking procedures as described in **Section 2.5.6**, **Access to Directories** do not apply. They can make new directories anywhere in the system filestore by calling *mkdir*( ), and remove them by calling *rmdir*( ) or *unlink*( ). (Normal users can only make or remove directory objects in directories to which they have write permission.) Take care not to remove a non-empty directory, because its contents may become inaccessible if there are no other links to them. Inaccessible objects can be recovered by executing a file-system integrity checking utility such as *fsck*.

2.7.3 Protected Subsystem

Processes with appropriate privileges can create a protected subsystem (see **Section 6.5.5**, **Protected Subsystems**) to restrict the process to a portion of the file tree by using *chroot*.

2.7.4 Signals

Processes with appropriate privilege can send any valid signal to any existing process. They can use *kill*( ) to kill a process. They should do this only as a last resort to kill runaway processes.

2.7.5 Process Control

Any process can call *nice*( ) and *ulimit*( ) to further limit its scheduler priority and file sizes. A process with appropriate privilege can use these calls to remove its limitations. Done incorrectly or maliciously, this can deny service to other users. On some systems, a process with appropriate privilege can lock part or all of the process memory into the main memory and prevent swapping or paging on these portions.

2.7.6 Setuid

If a process with appropriate privilege calls *setuid*( ) or *setgid*( ), it changes the real as well as the effective UID / GID. This is exactly what utilities like *login* and */bin/su* do. (If the process no longer has appropriate privileges, it ceases to be privileged, so the change is irreversible.) For a privileged program to assume a different UID on a temporary basis, it must create a subprocess and have the subprocess call *setuid*( ).

2.7.7 Special Services

On some systems, a process with appropriate privilege can call utilities like *mount*( ) and *umount*( ) to mount and unmount file systems, and *acct* to activate accounting. Also appropriate privilege is sometimes required for a process to take the last process slot. This guarantees the administrator access, if needed, to kill a runaway process that has used *fork*( ) to create many subprocesses.

2.8 REASONABLENESS TESTS

Software security checks may be waived for processes with appropriate privilege. However, X/Open-compliant systems also test whether attempted operations are reasonable. Not even privileged processes can perform operations that appear to be unreasonable; for example,

- No process can gain write access to a directory.
- No process can call *fcntl*( ) to break a lock placed on a file, except the process that placed the lock. However, if that process malfunctions, a process with appropriate privilege can kill it. This releases the lock.
- If a file has no execute permission, the system infers that the file is not executable. No process can apply an *exec* function to such a file. However, a process with appropriate privilege can change a file's permissions and then try to execute it.

2.8.1 Precautions

A file's attributes are independent of its contents. Moreover, a program's attributes can make it both set-user-ID and modifiable. These facts concern system vendors, since any code in a privileged program may execute with special privileges. Different systems may deal with this problem in one or more of the following ways:

- The system may strip a program's set-user-ID or set-group-ID attributes any time the file is modified, any time its owner or group is changed, or on any copy made of the program file. The owner of the file can always use *chmod* to reapply privileges to the program.
- The *chmod* utility and the *chmod*( ) call may ignore attempts to give an object set-user-ID or set-group-ID attributes if the caller also specifies write permission for anyone other than the owner. Alternatively *chmod* may treat such combinations as an error.

This Guide sometimes advises caution that may be unnecessary if your system implements one or more of these precautions. You should find out what precautions your system takes.

Chapter 3

Security for Users

The security of an X/Open-compliant system requires action on the part of every user. This chapter discusses a typical system, where users are responsible for protecting their own data, and describes other security concerns and the measures you can take with regard to physical security, logging in, and protecting files and directories. It also gives specific guidelines for using certain utilities.

3.1 PHYSICAL SECURITY

As a user you are responsible for applying physical and other forms of security to the computer components that you own or control.

3.1.1 Terminals

When private data is displayed, do not let others view the screen. Make sure your office terminal does not face open windows and doors. Ideally, install it in such a way that you can identify visitors to your office before they are able to view your terminal.

On a computer that can detect loss of carrier or of the DTR (Data Terminal Ready) signal, software may end any user session when it detects such a loss. On such systems, turn your terminal off and on again before starting a session. This helps to ensure that every time you type your password or other sensitive data, you are using your terminal in a ''known'' state, and that you are using the authentic *login* program (instead of an imposter designed to capture your password).

3.1.2 Intelligent Terminals

Intelligent terminals, such as personal computers (PCs) can sometimes be programmed to transmit screen data to the host computer based on a keystroke or on a message received from the host. A penetrator might be able to use the *write* utility to send a message to your terminal and cause it to respond in ways unknown to you. Mail messages and other data originating from other users might produce hidden effects on the terminal.

If you have such a terminal, keep others from writing to it at any time. Unless you require direct communication with other terminals, use this command to block broadcasts to your terminal:

```
mesg n
```

Before viewing text from other users, pipe it through a filter that will remove invisible (control) characters. For example, you can type:

```
tr -d [ \000-\037] < data_from_other_user
```

3.1.3 Unattended Terminals

Always log out before leaving your terminal. To secure an intelligent terminal, either lock your office or take other steps to prevent its unauthorised use.

Do not over-estimate the time required for a security breach. For example, do not disregard these rules because you expect to be absent from your terminal for only a short time. When using automated techniques, a penetrator needs only a moment at an unattended terminal with your authorisations to activate a penetration program previously placed on the system.

3.1.4 Printers and Plotters

Sensitive data in printed form is no longer protected by passwords or by the requirement for computer knowledge. You need to take special precautions.

Never print sensitive data on remote printers. If you must use a shared printer, log off and go to the printer before your data is printed, so that you can control access to it.

The most secure alternative is to print sensitive data on a printer in a secured computer room where operators will personally hand the output to you. A reasonable alternative is to print the data on a personal printer in your office. Consult your administrator before attaching a local printer to your terminal. Policy may require such printers to have a separate cable to the computer. This prevents autonomous ‘‘local copy’’ operations and lets the computer keep audit records of the date, time and user requesting each printing.

Once you have the output, treat it with whatever security it requires. Your organisation may already specify the use of locked cabinets, rubbish bags for incineration or shredders for sensitive documents, whether or not they are computer-related.

3.1.5 Diskettes and Tapes

Most sensitive information in X/Open-compliant systems is stored centrally; the administrator determines suitable security procedures for it. Diskettes and backup tapes may sometimes be under the physical control of individual employees.

The best way to protect these media is to lock them away safely. Diskettes and tapes fit in typical office safes. For larger media, the administrator may provide or specify a physically secure storage location. The average office cabinet or desk does not have an adequate degree of protection for truly sensitive data.

Changing the form of your data prevents casual misappropriation. If your system supports the *crypt* utility, use it to produce an encrypted copy of your file. Encryption may not always be an effective way to protect valuable data from unauthorised disclosure. For example, the algorithm in *crypt* is known to be breakable. Depending on the threats you have identified for your system, you may wish to consider other encryption techniques, especially if you expect to be confronted by sufficiently skilled penetrators.

The preceding paragraphs discuss threats to confidentiality. In addition, the integrity of data and programs is questionable whenever a medium leaves a secure area. If you lend a medium, never reinstall it on your system without analysing its contents or initialising it.

3.1.6 Personal X/Open-Compliant Systems

X/Open-compliant systems are implemented on some personal computers (PCs).

These computers require effective physical security. You usually achieve this by locking your office after hours. When PCs are located in shared office space, it may be impractical to move the entire computer to a secure location. However, you can place diskettes and tapes in a safe. On many PCs, fixed disks, such as Winchester disks, can be removed and safely stored.

Fixed disks are otherwise not secure. A thief can analyse data on a fixed disk by reattaching the disk to another PC. Any data or programs on media that you do not lock should be considered unsafe.

3.1.7 PCs Used as Terminals

Programs on PCs communicate with X/Open-compliant systems as though the entire PC were a video terminal. Most programs can capture the interaction to disk, and can send a disk file as though the keyboard operator were typing it.

The authentic terminal emulator can thus create disk files that may contain passwords and session transcripts that would give a penetrator access to additional data. This information survives in the copy even after it has been deleted from the central X/Open-compliant system.

A malicious program on the PC that appears to be your terminal emulator may record your password and anything else you type, even faking an entire session without your knowledge.

You can solve this problem by locking all media that contain programs on which you rely.

3.2 PASSWORD

The most important security transition occurs when you log in. Typing your password authenticates you as a user. Your password is your most sensitive piece of information on the system; knowledge of it gives a penetrator access to your data and to anyone else's data that you are allowed to use. A penetrator may be able to ensure perpetual access to your information even after you have changed your password.

A crucial security responsibility of all users, therefore, is to protect the secrecy of passwords. You must follow these rules:

- Look at your screen before typing. Any time it asks you to *login:*, take the steps described in the previous section to verify that the program making the request is the authentic *login* utility. If there are people in your office when you log in, ask them to look away while you type your password. Management should set the tone for the organisation so that nobody interprets your request as signifying distrust.
- Never disclose your password to anyone. It identifies you personally and you can expect to be held responsible (rightly or wrongly) for whatever happens through the mis-use of your password.

  Do not even give your password to operators or the administrative staff. Anyone performing sensitive operations on the computer should possess sufficient authorisations of their own.

  Other users requiring access to your data should consult you or the administrator. This may prove inconvenient or time-consuming, but the alternative destroys any security by eliminating personal accountability for what happens on the computer.
- Do not make copies of your password. It should be held only in an encrypted form in the system's files and in your head. Never write your password down, nor store copies of it, even on a different computer. Above all, do not program a PC to automatically log in to your account.

  If you use several different computers, you should have a different password on each one. This keeps a successful penetration of one machine from turning into a penetration of an entire network.
- Do not use a dictionary word as your password. Programs such as spelling checkers have an on-line dictionary which a penetrator could use to write a high-speed penetration program. Of course, do not use your company's name, a product name or similar items as a password.
- Choose a password that is difficult to guess (even by those who know your family, hobbies, or personal interests). Do not base your password on easy-to-locate personal information, such as names and birthdates of you or your relatives, car licence plate, social security number, telephone numbers, etc.
- Do not choose a password that is so subtle that it defeats security. For example, one that is so hard to remember that you must write it down. Do not choose a password that is so hard to type that you must re-enter it frequently, increasing the chance of its being studied by a bystander.
- Passwords should be changed periodically. The minimum and maximum lifetime of a password is dependent on the threats to which the system is exposed and the

environment in which it is operated. The longer the lifetime of a password, the greater the chance that a penetrator, even with limited resources, might determine the information, especially if he is able to gain access to the encrypted user passwords. The shorter the lifetime of a password, the greater is the probability of users choosing bad passwords, forgetting them or writing them down. The choice of the lifetime period of a password must be part of every security policy.

- Never use a password that you find as an example in this or other texts, as they are likely to be found in the special dictionaries used by sophisticated ''password cracking'' programs.

3.2.1 How to Choose a Password

There are several ways to choose a password that is easy for you to remember and hard for another person or a program to re-create.

- Use an acronym. Convert a phrase that is easy to remember, by taking the first letter of each word. In this way, ''This Is My First Secure Password!'' becomes ''timfsp!''.
- Devise a translation table that converts a word by substitution of letters. You can write down the translation table and then choose a dictionary word for your password, using the table to scramble the word before you type it into your keyboard. This method depends greatly on the confidentiality of the translation table which should be kept in a secure place.
- Use a variety of symbols, numbers as well as letters, mixing upper- and lower-case letters. Choose a password that contains all or nearly all of the maximum number of characters the system allows. This increases the time an automated penetrator needs to try all possibilities, and removes short cuts such as trying combinations of numbers alone.

3.2.2 Login Anomalies

A change in your normal login procedure may have security implications for the computer. For example:

- Changes in *login*. If you see messages from *login* that surprise you, verify with the administrator or an operator that the change is authorised or known to the administration. This includes cases where your password no longer works.
- Changes in start-up. If you have a PC in your office, its initial (start-up) process should be totally under your control. If it displays messages you are unaccustomed to, someone may have installed a Trojan horse program designed to trick you into typing your password. Do not use a start-up procedure that erases the screen and thus removes any evidence of unexpected changes.
- Other anomalies that are basic enough to have security implications include cases where you have typed a line and the shell ignores it, and cases where your screen displays text that did not come from you or from any program you were running.

If any of the above occur, log out and switch off your terminal. If they occur during login, stop immediately. Do not type your password. Report the strange behaviour to the administrator and use a secure terminal to log in. Execute the local equivalent of the *passwd* utility from there to change your password.

3.3 DIRECTORY AND FILE SECURITY

3.3.1 Directory Hierarchies

A ''hierarchy'' is a directory and all subdirectories under it. Your home directory is the directory specified in the user database that is your working directory at the start of each session. The shell variable *HOME* refers to this directory. Usually, the hierarchy under *HOME* is the complete set of files for which you are responsible. The things you must protect are:

- *Availability* — You will be inconvenienced if someone can delete your data. If you are maintaining data that others use, such as a database, you must protect the objects to prevent the waste of others' time and effort. If your objects are deleted, you may personally have to arrange for their restoration.
- *Integrity* — You will be equally inconvenienced if someone can render your data useless by making unauthorised changes to it. Unauthorised changes which remain undetected may lead to wrong results and fatal decisions based on them.
- *Confidentiality* — You know best how much confidentiality your files need. For example, your directories may contain personal data, corporate data that could harm the organisation if it reached a competitor, or a specialised data base containing confidential data of customers or employees. Software licences, labour agreements, such as your own employment agreement, and laws, may require confidentiality for certain data.

Protect Files

Section 2.4, **Objects and Permissions** and **Section 2.5**, **Access Rules** describe permissions and their meaning when applied to files and directories. You invoke the *chmod* utility to change permissions. For example, if other users need to read and execute one of your files, you might type:

```
chmod go=rx filename
```

If other users have no reason to use one of your files, you might type:

```
chmod go-rwx filename
```

You can see the permissions on files and directories by using the long form of the *ls* utility:

```
ls -l directory_name
```

To control the permissions given to newly created files, set your *umask* command appropriately, as described in **Section 3.4.3**, **New Objects**.

Protect Directories

Personal files are typically stored in a directory that has

`drwx------` (700)

access. As an alternative, the permission

`drwx--x--x` (711)

lets other users search through the directory and refer to its files.

Wider access to directories should be avoided as it creates special security problems. Write access for other users to your directories allows them to place Trojan horses (see **Section 1.1.6**, **Types of Penetration**) or to replace your files (see **Section 3.3.2**, **Temporary Directories**). Never give other users write access to your home directory. Files there control the operation of your entire session.

Create Suitable Subdirectories

The single set of permissions in a directory applies to the entire directory. You can give the objects in the directory different permissions on reading, writing and execution, but you cannot give them different directory permissions (which control other users' ability to reference the objects).

In practice, the objects in a directory often have the same or similar permissions themselves, except that execute permission may be withheld from non-executable files, such as some text files. If you find that the objects in a directory have widely different permissions, you may be able to improve security by creating additional directories with different permissions to hold the different types of file.

3.3.2 Temporary Directories

Many utilities create temporary files in your working directory or in the system temporary directories **/tmp** or **/usr/tmp**. Generally, anyone can remove an object in a temporary directory. Objects typically are removed when the computer restarts, nightly, periodically by the administrator, or by the utility that created them. You can protect files that you own and files that a utility creates in a temporary directory, by giving them few permissions. However, the temporary directory itself has no protection; its permission is

`drwxrwxrwx`

This lets anyone create objects in, and remove objects from, a temporary directory, without your permission or knowledge.

Substitution

A penetrator can remove an object from a temporary directory and quickly create an object with the same name. This is almost as powerful as the ability to make any desired changes to the contents of the object.

You may be able to detect this change: usually it gives the object a different owner and creation date (but you cannot prevent substitutions from occurring once your files move into an unprotected directory). Scripts that use temporary directories to store data or

pass objects between programs usually do not check for substitutions. Therefore, they cannot assure the integrity of the data they handle.

Some programs use standard library routines that allow an environment variable, called *TMPDIR*, to specify where temporary files reside. Some programs have options (like the **directory** option in the *vi* editor) that let you explicitly specify where temporary files should reside. Use these variables and options where possible, specifying a temporary file in your home directory.

Window of Vulnerability

There is a security risk in applying protection to objects placed in an unprotected directory. In this situation other users are prevented from using your data but they are allowed to link to it with the *ln* utility.

If you later changed techniques and decided to move the files to a protected directory and remove protection from the files themselves, other users could use links to your files that they made before you changed your technique. They could both refer to and use your data. Before removing protection from such an object, use *ls -l* to see if there are any links to the object. For ordinary files, a link count greater than 1 indicates extra links to the object. If this is the case, do not remove the protection. First, make a copy of the object using the *cp* utility. Then apply the *chmod* utility to the copy to remove its protection. This has no effect on the protection of the original and you can remove it from your directory. Other users who have secretly made links to the original object retain those links, but are still unable to use the object itself. When they remove those links, the original ceases to exist.

3.3.3 Set-User-ID Programs

Section 2.6, **Set-User-ID Programs** describes the set-user-ID file attribute. (This discussion also applies to programs with set-group-ID.) A process executing a program from such a file changes its effective UID to that of the owner of the file. This gives the program user the rights that the owner has. For example, the owner may also own a database that the users of the program need. The set-user-ID program gives its users access to such data, usually in accordance with rules imposed by the program.

The saved set-user-ID and saved set-group-ID feature of X/Open-compliant systems permit a set-user-ID program to change its effective UID back and forth between the UIDs of the user and the program owner. A malicious program may abuse this feature for a Trojan horse attack. You should therefore mistrust all set-user-ID programs of unknown origin if they are not thoroughly tested.

There is typically no reason for general users to own set-user-ID programs. If your directories contain any such programs, you should know exactly what they do. You should have a way to quickly make sure that the set-user-ID program is the program you think it is. Failing any of this, remove the program or its set-user-ID attribute.

If you must own a set-user-ID program, use *chmod* to deny anyone else permission to read or write it. This keeps a penetrator from studying the program and determining how to use its set-user-ID status to breach security.

Keep the program in a separate directory, preferably write-protected from all other users. Monitor the program carefully; use *find* as described in **Section 3.3.4**, **Directory Analysis**

to guard against changes to it. If you notice that the program has lost its set-user-ID attribute, or you have to restore the program from tape or a non-secure location, examine the program thoroughly or rebuild it from source before you use *chmod* to reapply the set-user-ID attribute. This keeps you from applying set-user-ID to a program modified by a penetrator.

Copying a set-user-ID program using *cp* has security implications on some systems. (See **Section 3.5.6**, **Copying Objects**.)

3.3.4 Directory Analysis

Review the contents of these directories from time to time. Examine any object you do not recognise by its title. Preferably, change its title to better reflect its contents. Never execute an object whose identity is uncertain.

If the *ls* utility ever lists an object in your directory that you have no permission to read, be sure you know why the object is there. It may be a program planted there by a penetrator. If the penetrator gets brief use of your account, or if you execute the program yourself, the penetrator may get ongoing access to your data.

It weakens the security discipline if another user stores objects in one of your directories, even if the user's intention is harmless. That discipline is necessary because a malicious user could obtain the power to damage your files by doing the same thing.

Using find

The *find* utility searches a hierarchy. You can make it print the names of files that meet certain tests. A command that begins with *find $HOME* searches your complete hierarchy.

Here are some ways to use *find*:

- Check for inadequate protection. This command prints the names of any objects in your hierarchy that a general user can modify:

  ```
  find $HOME -perm -002 -print
  ```

  There should typically be no such object anywhere in your hierarchy.

  In place of -002 in the example above, you can use -004 to find files general users can read, and -001 to find files general users can execute. You can use codes such as -020 to test group members' access to your data. (The ''-'' before the permission code is important. It says you are looking for files whose permissions include those you specified. For example, specifying -004 includes files that have 744.)

- Monitor changes to your data from time to time. For example, use this command once a week:

  ```
  find $HOME -mtime -7 -print
  ```

  This displays a list of all of your files that have been modified in the past 7 days. Take a moment to recall when and why you changed each file. If you cannot recall, examine the file and see if someone else has modified it.

- Check for set-user-ID/set-group-ID programs. **Section 3.3.3**, **Set-User-ID Programs** discusses the security risk of owning programs with the set-user-ID and set-group-ID

attributes. The long form of the *ls* directory listing identifies set-user-ID programs by listing **S** or **s**, instead of ''**-**'' or **x**, for the owner execute permission. The listing identifies set-group-ID programs by listing **S** or **s** as the group execute permission.

You can find any set-user-ID or set-group-ID program in your hierarchy by typing, respectively:

```
find $HOME '(' -perm -4000 -o -perm -2000 ')' -print
```

- Check for device special files, described in **Section 6.4**, **Machine Security**. User directories should not typically contain these. Type this command:

```
find $HOME '(' -type b -o -type c ')' -print
```

Using the output of any of these *find* utilities as input to *ls -laid*, you can see dangerous combinations of notable files—such as set-user-ID files that are also writable. For example:

```
ls -laid `find $HOME -mtime -7 -print`
```

3.3.5 Groups

The administrator may establish groups (as defined in **Section 2.2**, **Groups**) for the users in a certain department or for those working on a certain project. The X/Open compliant system records your group membership when you log in, based on data the administrator provides. Being a member of a group gives you greater permission to use one or more files of shared group data.

If you need access to several sets of project data, the administrator may register you as a member of several groups. One group will be your ''primary group'': the group to which you belong immediately after you log in.

If the supplementary GIDs mechanism is implemented (NGROUPS_MAX > 0), both these and your effective GID will be used to check group permissions when attempting to access a file. Alternatively, the optional *newgrp* utility lets you switch into a new group, to which you are authorised, at will. Always use the ''**-**'' option with *newgrp*. It sets up your process environment as though you had just logged in again.

For example, this command resets your environment and changes your group identifier (GID) to that of the ''electro'' group:

```
/bin/newgrp - electro
```

Group directories (directories containing shared objects for the group) have different permissions for group members and other users. A process's GID and supplementary GIDs determine whether the process qualifies for access to the directory. Group directories often have the permission:

```
drwxrwx---  (770)
```

If one user is responsible for applying changes to the objects in the directory, that user should create the directory while logged in to the relevant group. The user should give the directory the permission:

```
drwxr-x---  (750)
```

3.4 SECURE ENVIRONMENT

3.4.1 Profile Files

When you log in, the shell *sh* automatically executes the script named **$HOME/.profile**. This script typically specifies the rules that apply to your entire session. The **.profile** file determines the environment in which you will operate. You can edit **.profile** to make things happen automatically whenever you log in.

The contents of **.profile**, and the order in which they appear, are important to security. For example, if you let your environment include non-secure directories, you may unintentionally execute a malicious program written by a penetrator. This program may create a file, in the penetrator's directory or in a neutral or temporary directory, that is set-user-ID to your user identifier. The penetrator can use this program to acquire your powers, even if you later change your password or correct your **.profile** file. This security breach continues until the malicious file is destroyed or its attributes are changed.

The administrator typically places a **.profile** file in your home directory. Even so, you should understand what is in it and how it works.

Usually, the system wide profile file, **/etc/profile**, that is executed prior to your **.profile** file may contain all or some of the following items. If not your **.profile** files should contain, in this order:

1. A command to keep other users from sending messages to your terminal with *write*, as described in **Section 3.1.2**, **Intelligent Terminals**. Use the absolute pathname when calling the command to avoid the threat of a Trojan horse probably caused by an improper assignment of the system default environment.
2. A line that sets the *PATH* shell variable (see **Section 3.4.2**, **Search Path**).

   Putting this first ensures that all file references, in the rest of **.profile** and in your session, are well defined.
3. Lines that set other shell variables. These lines may influence the interpretation of the remainder of **.profile**. You may want to set *EXINIT* (see **Section 3.5.1**, **Editors**) and *MAILRC* (see **Section 3.5.2**, **Electronic Mail**).
4. An appropriate *umask* command (see **Section 3.4.3**, **New Objects**). A related function that might appear at this point is *ulimit*( ), which keeps malfunctioning programs from obtaining large amounts of disk space.
5. A site-dependent setup, provided by the administrator.
6. Finally, any commands you wish to take effect automatically on each login, such as commands to read your mail.

Here is an example of a **.profile** file:

```
# Disable broadcasts to this terminal:
/bin/mesg -n
# Set the search path and export it to all subprocesses:
PATH=/bin:/sys/bin:/usr/bin:$HOME/bin
export PATH
# Set other shell variables to influence the operation
#  of common utilities.
# If you have no start-up files,
#  specify no file, as shown below.
# Export these variables
#  to all subprocesses:
MAILRC=
export MAILRC
EXINIT=
export EXINIT
# Limit processes' resources:
umask 027
ulimit 50
# Site-dependent session initialisation might go here.
# Session start-up according to personal preference:
if mail -e
then
    echo "you have mail"
fi
```

Regard your **.profile** file as secret. Information in it, such as your search path, could help a penetrator determine how to make you execute a Trojan horse program. The permission to read the **.profile** file should be restricted to you alone, with

```
r--------
```

Change its permission to

```
rw-------
```

using the *chmod* utility, if you want to edit the file; change it back after editing it.

3.4.2 Search Path

Typing a filename into the shell utility constitutes a command to execute the file with that name as a program or a script. However, several directories may have files with the specified name. The file the shell utility selects normally does not depend on the working directory you have selected. Instead, it is dictated by the *PATH* variable.

The *PATH* variable is set by a shell utility that takes this form:

PATH = pathname:pathname:pathname...

To protect you from executing malicious programs, you should follow these rules when you specify your *PATH*:

- All pathnames should be absolute. Every directory in the *PATH* should be defined, have a specific identity, and be managed by a user who understands and applies the rules in this Guide for directories.
- No open directories should appear. If your search path includes an open or temporary directory such as **/usr/tmp**, a penetrator can trap you by placing a malicious program there with the same name as a program you are likely to execute.
- The working directory should not appear. You can change your working directory using the *cd* command, and then refer to files there without re-typing the pathname. (However, some of the working directories you select may contain programs you do not want to execute inadvertently. If you *cd* to another user's directory and your *PATH* includes the working directory ''.'', the other user has acquired the ability to corrupt your data using an automatic penetrator.) If required, you can still call programs in the working directory by prefixing the command name with **./**. You will soon become acquainted with this technique.

  Create a directory called **$HOME/bin** that will hold your private executable programs and scripts. Include this directory in your search path. Then, by copying or moving an executable object to **$HOME/bin**, you are able to invoke it by simply typing its name. Make sure that **$HOME/bin** is not writable by anyone else (see **Section 3.3**, **Directory and File Security**).
- Note the order of entries. This is the order in which the shell searches directories when you name a program to execute. System directories should appear first. If you inadvertently create an executable object called the *ls* utility, this order will keep you from executing it (until you rename it); any attempt to execute it will instead reference */s*.
- If the path includes another user's directory, be sure that user knows how and why to protect directories. Including another user's directory in your search path raises the risk that a penetration of that account will result in a trap set for you.

  A better approach is to explicitly copy or link the specific files you will need into your own directory, rather than setting your search path to implicitly obtain the other user's files.
- Analyse your *PATH* syntax. Any null entries in your path list specify your working directory. Furthermore, '':'' is a separator, it does not introduce. Therefore, if the specification to the right of ''='' begins or ends with '':'', or contains ''::'' anywhere, it has violated the previous rule.

You can check your *PATH* with a script like this:

```
if [ ‘echo ":$PATH:" | grep -c "::"‘ -gt 0 -o \
     ‘echo ":$PATH:" | grep -c ":\.:"‘ -gt 0 ]
    then
    echo "Error: PATH contains current directory"
elif [ ‘echo ":$PATH:" | grep -c ":[^/]"‘ -gt 0 ]
    then
    echo "Error: PATH contains a relative path"
else
    echo "PATH contains a proper search path"
fi
```

3.4.3 New Objects

The *umask* command helps you restrict the permissions of newly created objects. When you type *umask* followed by an octal number, the system keeps you from creating an object with any of the permissions specified by that number. The highest permission new objects can have is the complement of the *umask* number. For example, 027, a typical *umask* setting, specifies

```
----w-rwx
```

the permissions that would allow group write access and any access by other users. Therefore, *umask 027* keeps objects you create from receiving any of these permissions. Objects you create are limited to

```
rwxr-x---
```
(750)

which is the complement of 027.

- *umask 027* effects the rule described above
- *umask 067* further restricts group permission to execute-only.
- *umask 077* prevents any group or other access to your newly created objects

One of the above *umask* forms should be present in your **.profile** file.

Your *umask* selection specifies standard permissions for objects that you create, but it does not keep you from using *chmod* to change the permission of any object you own. It does not apply to objects you copy into your directory; the *cp* utility retains the copied object’s old permissions.

3.5 SPECIFIC UTILITIES

3.5.1 Editors

The **X/Open Portability Guide** describes screen editors called *vi* and *ex*. Two security comments apply to these editors (as well as to *edit* and *view* which are common on UNIX systems):

1. The editor creates a temporary file. This file assists recovery when the editing session ends abnormally. The file resides in the **/tmp** temporary directory, unless you have an **.exrc** setup file that uses the ''directory'' option to specify a different location. **Section 3.3.2**, **Temporary Directories** discusses the security problems inherent in temporary directories.

   If the system crashes during an editing session, most systems will use the temporary file to recover your edits when the system restarts. But if you lose contact with the editor for another reason (for example, if your terminal connection is broken), other users could substitute for the temporary file.

   Before you use *vi -r* to recover from a failure during an editing session, make a copy of the old file. Preserve the copy until you compare it with the restored file and verify that the changes correspond to your interrupted editing session. For example, you might type:

   ```
   $ cp file old_copy
   $ vi -r file
   :wq
   $ diff file old_copy
   ```

2. The second security concern for *vi* is user-specified initialisation. The *vi* editor takes initial commands from the *EXINIT* environment variable, if defined. If not, the editor looks for the **.exrc** file to specify the editing environment. **Section 3.4.2**, **Search Path** warns against using relative pathnames in your search path. Unfortunately, *vi* is designed with a relative search path for **.exrc**. It looks first in your working directory and then in your home directory. If you edit from someone else's directory or from a public directory, you may inadvertently use an **.exrc** file there whose contents are unknown to you. (Other utilities with **rc** files use this search path and therefore have the same security problems.)

   The *EXINIT* variable and **.exrc** file contain editing commands. Notably, the command :! executes a shell command from within the editor. Editing under control of someone else's *EXINIT* variable or **.exrc** file raises the possibility of executing a penetrator's program, such as a Trojan horse.

   You can always bypass the search for an **.exrc** file by defining *EXINIT* in your **.profile**. If you do not need special editor initialisation, set *EXINIT* to a null string:

   ```
   EXINIT=
   export EXINIT
   ```

If you use another editor with comparable functionalities, you should take analogous precautions.

3.5.2 Electronic Mail

The **X/Open Portability Guide** specifies two utilities for electronic mail: the simple *mail* utility, and the advanced *mailx* utility. The *mailx* utility uses a setup file called **.mailrc**. You can use the utilities interchangeably because they use the same format for mail files.

The *mail* utility has no method of user-specified initialisation. The *mailx* utility looks for an environment variable called *MAILRC*, analogous to the editor's *EXINIT* variable. You can set *MAILRC* to force *mailx* to use a specified file instead of **.mailrc**.

User-specified initialisation in *mailx* does not have the security problems of the editor. The *mailx* utility looks only in your home directory for **.mailrc**, not in the working directory. This reduces the chance that you will participate in a security breach by picking up the wrong **.mailrc**.

A potential security problem with any mail system is the authenticity of the sender. Both *mail* and *mailx* require a separate background process called a *daemon*. The daemon receives mail (as mail may arrive when you are not running *mail* or *mailx*). Some mail systems use a daemon to send mail. The daemon performs file operations based on the identity of the sender. However, some mail systems let a sender manually override the "From:" field in the header. Furthermore, if mail messages can originate from other machines, the authenticity of the sender may be in further doubt.

The integrity of the mail utilities, the daemons and all potential sources of mail, affect the security of electronic mail. Personal delivery, signed memoranda and telephone contact establish the sender's identity more reliably. An organisation should not use electronic mail to implement sensitive parts of its security policy, such as to assert that a new user is authorised.

If you use another mailer with comparable functionality to *mail* and *mailx*, you should take analogous precautions.

3.5.3 Network Communication

Anyone who uses *uucp* or similar network utilities should first have an idea of how secure the network is. Because *uucp* and related programs are not part of the kernel, most security checks are contained within those programs.

Section 6.6, **Communication** explains security problems you may encounter if your system is on a network with less secure systems, and security problems with the communication media between systems. If you cannot completely trust the communication path and the remote machine, you should consider programs like *uucp* to be inherently insecure. For example, assume that data you send using *uucp* could be intercepted. Likewise, examine any data you receive over the network.

3.5.4 Remote Sessions

The *cu* (call UNIX) utility lets you communicate with a remote computer. You use your terminal to interact with the remote computer as though the terminal were directly connected there. Before using *cu*, you must consider how secure the remote system is and how secure the communication medium between the systems is. Some of the issues discussed in the previous section apply to *cu*.

Using *cu* poses the same security problems as using a PC as a terminal (see **Section 3.1.7**, **PCs Used as Terminals**). You will be typing a password through *cu* to the remote computer, and you will be typically typing other commands that a penetrator might find useful to gain access to your data. Therefore:

- Be sure that the program you are using is the authentic *cu* program. Do not use a terminal that seems to already be running *cu*; exit and re-invoke *cu*.
- Do not devise an automatic login procedure, such as sending your remote password from a file on the local computer.
- If you are capturing the session transcript into a local file, begin the capture only after completing remote *login*. Capture only the data you need; avoid capturing the dialogue you used to obtain the data.

Suspended Session

Avoid leaving your terminal or using it for other things while a remote session is in progress. If your connection with the remote system is broken, immediately re-establish contact, check to see whether your first session produced any processes that are suspended, and kill those processes. Different X/Open-compliant systems may provide different ways for a penetrator to attach to a remote session that you neglect.

3.5.5 Backup and Restore

The *cpio* and *tar* utilities allow orderly backup and restoration of files between the file system and an archive on a medium, such as a tape or disk, that is not part of the file system.

On multi-user X/Open-compliant systems, the administrator or operations staff does backups. However, on personal X/Open-compliant systems, you may be responsible for backing up your own data.

Preparing a Backup

Typically you make a backup to ensure against a failure of the computer.

You should therefore plan for the possibility that you might later restore the files to a different computer. Each X/Open-compliant system has a different naming system for users, groups, UIDs, GIDs and file system structures. Therefore, use relative pathnames on your backup medium.

Imported media from another installation likewise have UIDs, GIDs and file system structures that do not relate to your system. Furthermore, on some systems, the *cpio* utility calls the *chown*() function to set the UID and GID of restored files based on what was on the tape. However, that UID/GID pair typically refers to another user altogether. The super-user may have to apply *chown*() to give the restored files the proper UID/GID for the system on which they now reside.

Security of Backup Medium

The resident files are as secure as the organisation's security policy. However, the archived files are not necessarily secure at all. They depend on the security that has been

applied to the archive medium. You should assume that restoration of files with *cpio* or *tar* is inherently insecure. In particular, the names and permissions on files on backup media are not necessarily an accurate description of the files' contents. If the medium is under the physical control of an individual, that person should perform the restoration, using their own user account and access restrictions.

Precautions

You should restore files to a temporary directory within your home hierarchy. You should give this directory permission

```
drwx------
```

preventing anyone else from using it, even if others will be able to use the files after you verify their identities and move them to their final destinations.

You should use the *t* and *v* options of *cpio* to view the contents of the archive medium before you restore any files. For example, you might type:

```
cpio -itcv </dev/sctmtm0
```

Note any special files, and files with permissions that are too liberal, or special attributes such as set-user-ID. Use *chmod* to change attributes if this is warranted.

Here is an example where you restore the contents of a directory named in the variable OLDDIR to a temporary directory (here called **readtape**) and analyse the result before actually restoring the files:

```
# The following special filename is assumed to
# reference an archive device with automatic
# rewind on close
#
RTP=${RTP:-/dev/rtp}

# Exit if the cpio archive contains absolute paths.
# (This example does not use the v option, since X/Open
# has not defined the output format when the
# combination cpio -tv is used.)
#
cpio -itc <$RTP 2>/dev/null | grep '^/' >/dev/null
if [ $? -eq 0 ]
then
  echo Archive contains absolute paths.
  exit 2
fi
```

```
#
# read archive into temporary directory.
#
umask 077
mkdir readtape
cd readtape
cpio -icvaml <$RTP

#
# Check for special files, FIFOs and files with
# the SETUID, SETGID or sticky bit set, and report.
#
echo The following files are character, block, or FIFO
echo special files or have setuid, setgid, or sticky
echo bit set

#
# The check for special files is only necessary if
# the script is run by a super-user
#
find . \( -type c -o -type b -o -type p -o \
      -perm -04000 -o -perm -02000 -o -perm \
      -01000 \) -print | grep '.*'
if [ $? -eq 0 ]
then
  echo Warning: Tape contains files with special modes.
  echo Press RETURN to continue.
  read x
fi

#
# Compare with original directory
#
dircmp -d $OLDDIR .

#
# Move to true place and remove temporary directory,
#
find . -depth -print | cpio -pcdvlamu $OLDDIR
cd ..
rm -rf readtape
```

Even when new files are extracted, consider them insecure until you fully analyse code and data.

tar

The *tar* utility cannot archive special files as the *cpio* utility can. This simplifies the auditing before restoring files. Use a command like this to view the contents of the archive medium before restoring any files:

```
tar tfv /dev/sctmtm0
```

To restore files with *tar*, use a procedure comparable to the example shown for *cpio*. On the *tar* command, use the *xfvlmp* options.

Some systems support the *p* option. This restores each file to its original mode, as carried on the archive medium. If you notice files with set-user-ID, set-group-ID or the sticky bit, omit this option and use *chmod*, once you have verified the files and moved them to their final directory, to place appropriate permissions and attributes on them. If your system supports the *o* option to place all files under your ownership, you may use this option but you must closely examine files that did not previously belong to you. Use the technique described above to obtain a contents list for the archive from which to identify such files.

Physical Security

Take precautions to prevent a penetrator from writing to your backup medium after you mount it. If your installation has a program to request exclusive access to the drive, reserve access before your medium is physically mounted, and do not release the drive until your medium is physically unmounted. If your installation supports it, issue a command to unload your medium from the drive after backing up or restoring files. Retrieve your tape from the drive promptly after you have finished using it.

3.5.6 Copying Objects

The *cp* utility makes copies of objects. If you copy data to an existing object, the ownership and attributes of the destination object do not change, and there are no special security issues.

If you use *cp* to create an object, you are its owner.

However, the file's data and its other attributes may follow those of the source of the copy. On some systems, if you copy a set-user-ID program (see **Section 2.6**, **Set-User-ID Programs**), creating a new file, the new file is set-user-ID to your UID/GID. **Section 3.3.3**, **Set-User-ID Programs** explains why this may be a problem. In this case, after you have copied a set-user-ID file, use *chmod* to remove the set-user-ID attribute.

Your *umask* (see **Section 3.4.3**, **New Objects**) limits the permissions that new objects receive. The *cp* utility ignores your *umask*. In other words, by copying, you can place objects in your directories with permissions more liberal than those the system would let you apply to new objects. This is a good reason to periodically review all of your directories, as described in **Section 3.3.4**, **Directory Analysis**.

This discussion of the *cp* utility also applies to the *mv* utility, which ''moves'' an object. If the source and destination file are in the same file system, *mv* creates a new link to the existing object (as does *ln)* and removes the old link. This never affects the object's protection. However, when ''moving'' an object to a different file system, *mv* copies the object, as does *cp*. As described previously, this may require care when moving set-user-ID programs.

3.5.7 Deferred Scripts

The *at* utility lets you execute a script at a future time. Enqueuing a job for *at* essentially schedules an automatic login under your login name at a future time. The script may make assumptions about the future contents of certain stored objects. A penetrator who detects and analyses your *at* job knows that a login under your user name will occur at a certain time in the future. By studying the script, the penetrator may be able to make changes that trick your automatic login into giving the penetrator some of your power.

For sensitive operations, and for operations that depend on co-operation from other persons, such as tape operators, execute the script manually and watch the output on your terminal.

Do not use *at* for scripts you wish executed in the near future. If the system is busy, *cron* may not schedule your batch until the desired time tomorrow. Use *batch* instead. If the *ps* utility reveals that a process is not running as expected, cancel the job.

Capture the output (standard output and error output) from all deferred scripts. Either redirect it to a file, using a full pathname, or specify that the output be mailed to you. (On some systems, *cron* does this implicitly.)

Chapter 4

Security for Programmers

Every programmer of an X/Open-compliant system is also a user, and should read and follow the guidelines in **Chapter 3**, **Security for Users**. For example, programmers should protect passwords, apply proper permissions to their files and know how to check their computer environment for possible security problems.

This chapter discusses additional security-related topics of relevance to programmers. Some of this information, such as the need to take precautions when using shared data structures, applies to any time-sharing system, but the technical information and all examples are specific to X/Open-compliant systems.

Special authorisations are not needed to write programs on an X/Open-compliant system, but the programming in itself may have security implications, as in the following situations:

- Where you are maintaining a utility that has inherent security implications, such as *login*.
- Where you are writing or maintaining a program that will require special privileges when completed and installed for general use.
- Where you are writing or maintaining a program in which others will invest time and effort.

4.1 PROGRAMMING MANAGEMENT

Good programming management is obviously important, because it produces more efficient results from the programming staff. It is also important, however, when developing or maintaining security programs, because of the impact of such programs on other users. The following principles should therefore be applied to all programming projects:

- Assignment of personal responsibility. One way to apply this general principle in software development is to assign an owner to each software module. Changes to the module, however minor, should be cleared with, and preferably applied by, the module's owner. **Section 3.3**, **Directory and File Security** describes how to use permissions to enforce this rule.
- Encouragement of structured programming. The objective of structured programming is to produce code that is easy to read and maintain. Structured programming saves development time and improves maintainability without detracting from the final product's size or speed. This is especially important for sensitive programs.
- Review of program design. A programmer assigned to revise a program does not necessarily understand all parts of the program, how it fits together or aspects of the program that are important to maintaining security. Misunderstandings can occur even if the program is well documented. Group review of program changes can sometimes identify potential security problems, and should catch any potential program malfunctions and make it less likely that departures will be made from the original design philosophy.
- Provide reproducible test suites and document them. If possible make these tests run without user interaction and give a clear indication of success or failure. Use this technique not only for the whole product but also for each subsystem. In case of abnormal behaviour by corrupt programs, there is then an opportunity to check the components.

The following aspects of good programming management take advantage of specific X/Open features:

- Use of the Source Code Control System (SCCS). The SCCS system helps manage software projects by assigning ownership of individual modules to programmers making changes. It provides a change history that makes it easy to see the essence of each revision of the source code. If a security breach appears to have involved deliberate changes to security programs, SCCS may help to identify the culprit. However, this will not work for unauthorised changes.
- Control of the application of privileges. Create a non-critical environment for programs under development, such as a protected subsystem (see **Section 6.5.5**, **Protected Subsystems**).

  Establish a policy that programs must be written in such a way that they can be tested on non-critical data, without privileges such as the set-user-ID or set-group-ID attributes. Apply those attributes only after the code has been reviewed and all affected departments are satisfied that the new programs maintain security.

4.2 PROGRAMMING GUIDELINES

The following guidelines are useful in all programs that use shared resources. They address security issues because a runaway process, especially a privileged one, can deny service to other users by requesting an extreme amount of resources.

4.2.1 Analyse All Return Codes

Virtually all function calls described in the **X/Open Portability Guide**, **Issue 3**, **Volume 2**, **XSI System Interface and Headers** to **Volume 4**, **Programming Languages** inclusive return numerical result codes. Programs should verify that each service request reports success. In the case of failure, the programmer should ask:

- Should this failure be reported to the user?
- Does this failure affect the security of the program or of the system? Should it be recorded in a log?
- Can the program proceed? Should the program retry the failed operation?

Often the answers to these questions are different for different types of error, in which case the program must analyse the numerical value of the result code and handle each case separately.

There are several notable causes of program failure:

- **Section 4.4**, **Privileged Programs** describes error returns from system calls that suggest a security breach, and suggests ways in which the program should respond.
- Privileged functions can execute calls such as *chroot*( ), which other functions cannot execute successfully. However, the function or a copy of it may be executed without its privileges. The program should detect this situation and exit cleanly if it is not able to perform the user's request.
- The *fork*( ) call can fail if there is insufficient space in the memory or in the process table to create another process. The result code from *fork*( ) distinguishes the parent process (result > 0) from the child process (result = 0). Programs often fail to test for a negative result code, which indicates failure. Here is an incorrect example:

```
/* INCORRECT EXAMPLE */
int pid;
...
if ((pid = fork()) == 0) {
  child_process_code;
  exit(0);
}
parent_process_code;
...
```

In this example, the parent process executes `parent_process_code` regardless of whether *fork*( ) succeeded. The code should be structured like this:

```
/* CORRECT EXAMPLE */
int pid;
...
switch (pid = fork()) {
  case 0:
    child_process_code;
    exit(0);
    break;
  case -1:
    report_error;
    break;
  default:
    parent_code;
    break;
}
```

- Programs should even check for error cases that seem impossible, since changes over the life of the program may make those errors occur.
- Programs can test the global variable *errno* to obtain additional information on why certain calls have failed. This can help the program decide how to respond, or produce a more helpful diagnostic message.

  User programs should not modify *errno* (except in the manner described by the **X/Open Portability Guide Issue 3**). This ensures that the text obtained from *perror*() or *strerror*() is valid and reasonable.

4.2.2 Write Portable Code

The **X/Open Portability Guide** defines many constants to account for different implementations. Programs should always use the symbolic names for these constants instead of coding the actual value that is correct for their current implementation. Where a program's constants could change in different implementations as a function of one or more symbolic constants, you should write the appropriate formula, not code the value that works at the particular time.

For example, the constant {OPEN_MAX} is defined in **<limits.h>** as the maximum number of open files. At the start of a program that could run as a *fork*() process, you might want to ensure that all files are closed. You might write:

```
for (fd = 0; fd < OPEN_MAX; fd++)
  (void) close(fd);
```

The typical value for {OPEN_MAX} is 16. But if you coded 16 into your program instead of {OPEN_MAX}, your program might fail to close all files if run on an X/Open-compliant system where a process could have twenty files open simultaneously.

4.2.3 Examine Your Environment

If your program makes assumptions (about the language, compiler, processor or instruction set) that may differ among implementations, start your program with code

that tests your assumptions and reports failure if they are not met.

Any assumptions your program makes about its caller should also be verified. For example, the program should test signals, examine the *umask*( ) under which it is running and verify any files that the caller should have opened.

Give up unneeded privileges. Programs that do not need set-user-ID privileges may have them anyway if someone applies the set-user-ID attribute to the executable file or to a copy of it. Such a program should start by explicitly giving up privileges. For instance, call *setuid*( ) to set your effective UID back to your real UID:

```
retcode = setuid (getuid());
```

4.2.4 Follow Programming Discipline

Remember that you are writing a program that will be maintained by other people, perhaps after you have left the organisation. Their ability to maintain your work, and the organisation's ability to assure that your product preserves the security of the system, depends on your communication. Typically, the most reliable method of communication is to document your design, include explanatory comments liberally throughout your program, and write the program so that it can accommodate future changes.

Not only is programming discipline necessary for the initial version of the program but it must also be maintained throughout the program's lifetime. Hence for every change to the code, corresponding changes should be made to the design documentation, the code comments and the change history.

4.2.5 Define Appropriate File Access Rights

Be sure to give the proper access rights to files created by your programs. You may not want to rely on the process's file mode creation mask, nor on the user ID and group ID of the process. If necessary, call *chmod*(), *chown*() or *chgrp*() explicitly to set the appropriate access rights. Remember that even for programs that are restricted to a group of users, you may not make assumptions about the group ID because of the supplementary group mechanism of X/Open-compliant systems. You should pay the same attention to files you maintain for recovery and backup purposes and not give wider access rights to them than you give to the data files themselves.

4.3 MULTI-TASKING GUIDELINES

4.3.1 File Access

Whenever you write or modify a program, you must consider the possibility that two users may run the program at the same time. If this makes it possible that two processes may use the same file, you must ensure that they do so in an orderly way.

Static Files

Most public files are not meant to change. Some, like program image files, never change once they are installed. Programs do not need to anticipate competition for these files, since the exact access sequence does not matter when processes are simply reading the files.

Some application packages are designed so that a file containing application data is read-only, at least during multi-tasking. For example, a transaction package may perform posting functions as a ''batch'' at a time of day when transaction processing is not occurring. This eliminates problems with simultaneous writes to certain files, if all programmers obey the plan.

File Locking

If a process is going to modify a file's contents, other processes must neither read nor write to the file at the same time without precautions. A process independently reading the file might read a version of the file that was only partially updated by the modifying process and therefore contain inconsistent data, which could lead to a malfunction. Two processes trying to write to the same file at the same time without precautions could interfere with each other. Each process's in-memory copy of the disk data might not contain each other's changes. The second process that wrote to disk might therefore undo some of the changes that the first process applied to the file.

A simple way to prevent simultaneous access to a file is for each modifier to make a copy of the file and to apply the modification to the copy. (Other potential modifiers, detecting that a copy existed, must delay their attempt to modify the file.) Once the modifier has finished the change to the file, it uses the *link*( ) call to make the old name point to the new data. The change seems instantaneous to all other processes.

If the modifier places the copy of the file in a temporary directory, programmers and users should read the warnings in **Section 3.3.2**, **Temporary Directories**.

File locking by changing links is adequate for small files whose contents undergo changes infrequently.

Record Locking

An X/Open-compliant system supports locking of individual records in files. This is the preferred method of preventing simultaneous writes to files. The *fcntl*( ) system call gives a process exclusive access to a single record in a file, as in the following example. All other processes lose their ability to read or write to the locked record, but are still able to use the rest of the file normally.

```
/*
 * Example of file locking with fcntl
 */
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
main(argc,argv)
int argc;
char *argv[];
{
  int      fd;
  struct flock lck;

  if (argc != 2) {
    fprintf(stderr,
      "Usage: %s filename \n", argv[0]);
    exit(1);
  }

  /* open file for reading */
  if ((fd = open(argv[argc - 1], O_RDONLY)) == -1) {
    fprintf(stderr, "%s: Cannot open %s \n",
      argv[0], argv[argc - 1]);
        }

  /* set up location and size of segment lock */
  /* offset from start of file */
  lck.l_whence = SEEK_SET;
  lck.l_start = (off_t) 2;
  lck.l_len = (off_t) 10;

  /* lock the segment */
  lck.l_type = F_RDLCK;
  fcntl(fd,F_SETLK,&lck);

  /* read the segment */
  /* unlock the segment */
  lck.l_type = F_UNLCK;
  fcntl(fd,F_SETLK,&lck);
}
```

Recovery from Locks

Applying locks to files has security implications in that if your program fails with locks outstanding, it can deny service to many users. Programs that lock files should check exit status codes from any system call and ensure that they remove their locks as soon as they are no longer needed.

If a program hangs with locks outstanding, the locks are easy to clear if the program uses temporary files. You can manually delete the copied file or change links. If the program locks individual records, you must find the failed process and issue *kill*. This releases any records that the process locked.

Deadlocks

Security-related programs must be especially careful to avoid deadlocks. A recommended technique is for a program to obtain all locks prior to performing its real function.

In the case of a deadlock occurring, *kill*() can recover the situation. One method of detecting and managing deadlocks is for a program to create a subprocess in which to manage the locks and perform its critical function. The parent, meanwhile, imposes a timeout interval on the subprocess so as to detect a deadlock.

4.3.2 Subprocesses

The *fork*( ) call lets a program create another process. In this way the program can specify multiple actions that take place in parallel.

Shared Open Files

To communicate with a subprocess, the parent often opens files. The subprocess can use these files. A process that is about to create a subprocess with *fork*( ) should explicitly close all files it has opened except those that the subprocess will need. This is especially necessary for privileged programs where the subprocess may release its privileges.

Opening files is also a way to communicate with different programs that a single process chains to by calling one of the *exec* functions. The successor program has continued access to the files that its predecessor opened. Programs should therefore close all unrelated files before calling one of the *exec* functions. An alternative is to use *fcntl*( ) to set the FD_CLOEXEC flag on these files

Environment

When a process calls *fork*( ), the subprocess receives an exact copy of the parent's environment. If a program chains to another program by calling one of the *exec* functions, it may specify a different environment for the target program. **Section 3.4**, **Secure Environment** describes the importance of a secure environment. For example, the *PATH* variable, if set incorrectly, can cause a process that executes the shell, *sh*, inadvertently to obtain files from an improper directory.

Signals

Subprocesses inherit the signals set by the parent process before calling *fork*( ). Each process should examine its signals. Subprocesses that implement background activities often explicitly specify SIG_IGN for some signals, so that they will not respond to real-time events such as keystrokes.

The following is a useful example of code to reset a program's environment:

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
main (argc, argv)
int argc;
char *argv[];
{
  mode_t old_mask;
  int sig, fd, max_open;
  struct sigaction *sig_state;
  sigset_t set;

  /*
   * Reset standard error to ensure the arrival
   * of error messages at the terminal.
   *
   * Similar action may be necessary for standard
   * in and standard out.
   */

   freopen ("/dev/tty", "w", stderr);
  /*
   * Exit if unable to determine
   * highest signal number
   */
  #ifndef NSIG
  fprintf (stderr, "NSIG not defined\n");
  exit (1);

  #else
  /*
   * Allocate space for signal structures
   */
  sig_state =
    malloc (sizeof (struct sigaction) * NSIG);
  if (sig_state == NULL) {
      fprintf (stderr, "malloc fails\n");
      exit (2);
  }
```

```
    /*
     * Obtain and save all signal states
     */
    for (sig = 1; sig < NSIG; sig++)
      if (sigaction (sig, NULL,
        sig_state + sig) == -1) {
            fprintf (stderr, "invalid signal\n");
            exit (3);
        }

    /*
     * Close all open files except for
     * ones expected to be open.
     */
    max_open = (int) sysconf (_SC_OPEN_MAX);
    for (fd = 0; fd < max_open; fd++)
      if ((fd != fileno(stdin)) &&
          (fd != fileno(stdout)) &&
          (fd != fileno(stderr))) {
          if (close(fd)) {
              fprintf (stderr, "close fails\n");
              exit (4);
          }
      }

    /*
     * Save the old mask and specify a more
     * secure mask.
     */
    old_mask = umask(~(S_IRWXG | S_IRWXO));

    /*
     * Remainder of program follows.
     */
    #endif
}
```

4.3.3 Unrelated Processes

Signals

A process can send a signal to another process (see **Section 2.5.2**, **Access to Processes**). The **X/Open Portability Guide** specifies a set of ‘‘user-defined’’ signals, which should be used in preference to the signals reserved for bus, keyboard, protection or other events for inter-process communication.

Signals effect one-way inter-process communication. The operating system keeps track of signals even while the receiving process is dormant; it may or may not keep track of multiple occurrences of the same signal. Unless the receiving process acknowledges the signal by sending a signal back to the first process, multiple requests for service may

produce only a single delivery of the service. A signal might even be completely ignored because the target process terminated or disabled the signal. In these cases, a combination of signals and messages should be used. Messaging can be implemented using X/Open IPC mechanisms, if supported, or FIFOs. The sender should identify itself by a message specifying the requested service, which it notifies to the receiver by means of a signal.

The **X/Open Portability Guide**, **Issue 3**, **Volume 3**, **XSI Supplementary Definitions** defines shared memory, semaphores and message queues. These are optional techniques that effect inter-process communication. They have a unique identifier, owner, group, permissions and attributes, just like files. You can control access to IPC objects in the same way as for files. These optional IPC objects may produce more reliable inter-process communication than that produced by reliance on signals alone.

FIFO Special Files

All X/Open-compliant systems support the ''FIFO special file'' mechanism for inter-process communication. You call *mkfifo*( ) to create a FIFO special file for reading or writing, give it a pathname, and specify its attributes and permissions. You should give FIFOs no greater permissions than are needed to effect the communication.

Temporary Files

In some situations you need to ensure that unrelated processes do *not* have access to a program's data, such as temporary files. A named file is addressable and accessible to any process with sufficient authorisation. The usual way to effect temporary files on an X/Open-compliant system is with *tmpfile*( ). Although the temporary file appears in no directory, it remains in existence for the duration of the process, because the process retains a reference to the file. The process can pass the file to subprocesses through *fork*( ) or to other programs through one of the *exec* functions. When the last reference is discarded, either explicitly by a call to *close*() or implicitly by a process exit, the temporary file ceases to exist.

4.4 PRIVILEGED PROGRAMS

Section 2.6, **Set-User-ID Programs** and **Section 2.7**, **Privileges** define the set-user-ID and set-group-ID attributes and privileged programs. In X/Open-compliant systems, a privileged program typically obtains more power than it needs in order to do its job. For this reason, privileged programs should take special precautions, and should implement the principle of least privilege themselves by reducing their power when appropriate.

- Identify the user. As your program begins, the set-user-ID attribute may have changed the process's effective UID. The program's behaviour, and its ability to perform sensitive operations, depends on the effective UID. However, the process still identifies (as the real UID) the identity of the user that invoked the program. The *getuid*( ) call obtains this information.
- Consider using an ''authorisation file'': i.e., a file that lists the users who are entitled to use the privileged program. In this way the program is written so that it exits without performing service if invoked by a process whose real UID is not found in the authorisation file.

  The *access*( ) system call checks file access based on the real UID. That is, it tells you if the user could have gained access to a specified object without the privileges it gained from your program.

  For example, a program might require files in a directory that the typical user has no permission to read. The programmer finds it acceptable to give the user access to these files under the control of the program, so the program is given the set-user-ID/set-group-ID attributes. However, the user may specify other files, but the programmer wants to keep the user from taking advantage of the program's set-user-ID/set-group-ID powers. In this situation the *access*( ) system call confirms that the user has independent permission to use a specified file.
- Log all uses. The privileged program should create a log file to record all uses of the program. Take advantage of the program's privileges and locate the log file in a directory that the user could not otherwise gain access to. Ensure that the log file has no permissions for users other than the owner; for example, give it

  ```
  -rw-------
  ```

  Record at least the date and time of use, the user command and the user's real UID/GID. (The effective UID is typically a constant because the program is set-user-ID.)

  Always log any abnormality or apparent security violation, even unsuccessful penetration attempts. Logging can be achieved by sending *mail*. **Section 6.5.4**, **Contents of Objects** gives a useful example of this for shell scripts.
- Handle security errors intelligently. If the program detects that its current user is not authorised, it should always log the incident.
- Deny information to penetrators. If your program's user authentication tests fail, presume that the user is trying to penetrate the system and design your program to disclose as little information as possible.

  When producing numerical return status codes, one number should cover all types of security violations. Alternatively, a security violation should be covered by the same

result code that applies to attempts to use an unimplemented feature.

Never display a diagnostic message that discloses the nature of the security violation. For instance, never list the name of the authorisation file the user's UID did not appear in. Always test for a security violation before trying to do any part of the prohibited operation. Do not provide information about whether the operation would have succeeded for an authorised user.

You may elect to program a small delay before issuing the diagnostic message. This hinders programmed attempts to penetrate your program by systematically trying a variety of inputs.

- Give the administrator flexibility. Have your program use command line arguments, environment variables or configuration files to stipulate anything that an administrator might need to specify to ensure your program's security. This includes script names and default directories. Particular names hard-coded into your program may not be secure areas at some installations.
- Give up privileges as soon as possible. **Section 4.2**, **Programming Guidelines** suggests starting a program with a form of the *setuid*( ) call that forfeits unneeded privileges granted by a program's set-user-ID attribute. You can also call *setuid*( ) later in the program to give up privileges after taking steps that required privileges.
- Give up indirect privileges. Whether your program exits successfully or unsuccessfully, always delete temporary files. If a penetrator finds a program that produces suitable files, even an unsuccessful use of that program may be sufficient to empower another program to breach security. If temporary files contain security relevant or confidential information, you might need to catch signals to ensure they are deleted should the process be abnormally terminated.

  You should close files immediately after using them, especially if you used special privileges to gain access to them. Doing this, rather than relying on the automatic mechanism that *fcntl*( ) provides, makes it more likely that the files will be closed if the program ends abnormally in ways that you do not anticipate.
- Use ''privilege bracketing'' whenever possible. An X/Open-compliant system supports a saved set-user-ID and saved set-group-ID which allows set-user-ID and set-group-ID programs to change their effective UID and GID. Privileges should be removed immediately after program startup. When privileges are needed, surround that code with calls of *setuid*() and *setgid*() as appropriate. Privilege bracketing in this manner reduces the chance of privilege abuse.

  If the privileged program allows escapes to other programs (e.g., *sh*), it must first release its privileges.
- Consider having the program use a protected subsystem (see **Section 6.5.5**, **Protected Subsystems**). This makes the program incapable of being applied to the system at large.

4.5 SPECIAL CASES

4.5.1 Shell Scripts

The permission scheme in the **X/Open Portability Guide** lets you enable a class of user to execute a program but not determine its contents. You do not have this flexibility when the program is a shell script. To execute a shell script, a user must have both read and execute permissions. This lets the user view the exact procedure the shell script uses.

Programs that are essential to security should be written in a language such as C and made available to general users only with execute permission to the executable file. A procedure implemented as a script cannot be protected adequately by including authentication and logging in the script. Any penetrator executing the script could make a copy of it and remove those features. Also bear in mind that a shell script should never have the set-user-ID or set-group-ID permissions set.

4.5.2 Daemons

A daemon is a background process that performs some service independently of any user. Typically, a user requests action from the daemon. The daemon may take the action later; the user may not then be present at a terminal.

The user typically invokes a ‘‘helper program’’ to communicate with the daemon. The helper specifies what action the daemon should take by sending specific files to the daemon or by using various inter-process communication techniques. For example, *at* is a helper program for the *cron* daemon.

Follow these guidelines when writing daemons:

- Study your use of privilege. The helper program typically has to be a privileged program to communicate with the daemon, but the daemon may not have to be privileged. The daemon may be owned by a pseudo-user and perform useful work by manipulating objects owned by the same pseudo-user, to which users could not gain access themselves.
- Structure the daemon’s work. After the daemon initialises, its activity should be divided into discrete tasks that carry out a separate user request. For example, one pass through the daemon’s main loop might correspond to a single use of the helper program.
- Verify the user’s identity. Unlike a privileged program, a daemon cannot call *getuid*( ) to determine the real UID of the process that is requesting service. The helper program, as a privileged program, should verify the real UID of its requester and include that in its request to the daemon. If the helper requests service by passing a file to the daemon, the daemon should do more than just examine the file’s owner in order to determine correct identity. In some systems, any user can call *chown*( ) to change a file’s owner (to give it away), and by using this technique a penetrator could convince the daemon that a request came from someone else.
- Use resources carefully. Always bear in mind that the daemon usually continues running indefinitely. For example, release terminals as soon as possible or you may delay a new session at that terminal.

4.5.3 Indexed Sequential Access Method (ISAM)

The **X/Open Portability Guide**, **Issue 3**, **Volume 5**, **Data Management** describes an Indexed Sequential Access Method (ISAM) for managing data files. A typical X/Open-compliant system implements ISAM through library routines instead of system calls. System calls could impose security restrictions based on the user and the purpose of the call. Library routines have no special ability to restrict certain callers.

If you are using ISAM routines to implement a sensitive database, you must use other protection techniques that the operating system interfaces provide. For example, make sure that your data access technique is the only one that can give a general user access to the sensitive files.

A typical way to do this is to apply protection to the data files. For example, there could be a pseudo-user that is the owner of the data files, or the files could belong to a special group to which no real user of the system belongs. Permissions on the data files prevent general users from any access to them.

Once the files are protected in this way, users must invoke a program with the set-user-ID or set-group-ID attribute to gain access to the files. This program must authenticate the user's identity and the user's authority to perform the desired operation on the files. Dependent on the data, users decide whether the contents of files should be closed.

In situations where many users need to update files simultaneously, the simplest way of implementing an ISAM application may be to write a daemon (see **Section 4.5.2**, **Daemons**) and have general users invoke a helper program to ask the daemon to read or write to the file.

4.5.4 Structured Query Language (SQL)

Structured Query Language (SQL) is a language and a set of corresponding routines that implement relational databases. On many X/Open-compliant systems, SQL is implemented using ISAM (described above) and this means that SQL raises the same security issues as ISAM.

File Security

SQL information is stored in files, which must be accessible to general users unless you use an access scheme via a privileged program.

Imprecision of Authorisation

As SQL is defined, the user who creates a relation (a table) owns it. That user can selectively grant and revoke access to other users but a user thus authorised has access to the entire table. This is because each table is a separate file.

You may often need to restrict certain users by row or by column. For instance, you may want to deny some users information on certain records (certain persons, data points, accounts, or so on). Or you may want to deny some users certain data fields pertaining to all records in the database (for example, the salary or telephone-number field). SQL lets you do this, but only by your designing the database as separate tables; this reduces the efficiency of such a database.

Inference

Inference means a user's ability to gain information by indirect query. It can be a security problem in any database system. Although a database may restrict a user from asking for certain information, the user can infer the desired information by asking related questions that are not restricted.

Other than revoking a user's access to a given relation, you can make inference less likely by forcing query sizes to be large, or by randomly restricting the data released with each query. Each approach has practical disadvantages.

Chapter 5

Managing Security

To establish a security policy for a computer system based on an assessment of the security risk, an administrator should consider:

- the value of the system and its contents
- the potential threats to the system
- the system's vulnerability to the threats

A good security policy sets out cost-effective countermeasures against the risks.

This chapter considers the different aspects of a system's value and the threat assessment. It then describes the many ways in which the administrator establishes and maintains a secure X/Open-compliant system.

Related Information

Chapter 6, **Administrative Procedures** gives technical details of specific operations done by the administrator. **Section 4.1**, **Programming Management** gives advice for programming managers.

5.1 ESTABLISHING SECURITY

5.1.1 Occasions for Re-evaluation

A common situation that causes an organisation to study security arises where management or the administrator believes that a system is insufficiently secure. Dissatisfaction with the previous state may be the result of a new factor, such as:

- A new computer system, for which security policy has not been devised.
- A new administrator, either newly hired or assigned from other duties, who sees a need for increased security.
- Renewed management interest in potential security problems, perhaps based on events within the company, stories heard from other managers or news stories. Security often undergoes complete re-evaluation at this time.

Re-evaluation may take place if:

- The company's security strategy enforces regular re-evaluations as is required by some standards like ISO 9000.
- The legal situation changes (e.g., new laws to enforce data security, the system begins to use data which is subject to existing laws).

Section 1.1, **Components of Security** explains that the computer system has a value and the goal of security is to provide assurances that the integrity of the system will be preserved. If the organisation responds to the need for greater security by simply increasing funding, specific problems may be solved but the funds may not be spent optimally. A better approach is to evaluate the entire security policy to see whether it meets the goal of security. In other words, instead of asking, ''How much more security does this computer need?'', the organisation should ask, ''How much security, and in what forms, will ensure the preservation of the system's integrity?''

Before you can answer that question, you need to take an inventory of the system's value.

5.1.2 Value Assessment Inventory

The following aspects of a computer system that needs to be protected by the security policy should be considered when assessing its value.

Physical Equipment

Physical equipment has a readily quantifiable value that the organisation should relate to the level of security required.

Physical security is required to protect physical assets (see **Section 1.3.1**, **Physical Security**).

Investment of Time

Purchased software can be replaced or purchased anew. But proprietary software, including the installation's custom administration programs, the company's custom applications and user data, can only be recovered by repetitious work on the part of the employees. The value of this software to the organisation can be quantified by estimating

how much it would cost to replace the software.

Frequent, secure backups and off-site storage can protect the products of time and effort already invested from all dangers except deliberate attack. To protect against deliberate attack, adequate software security is required.

Availability

The denial of service by a computer could have a quantifiable effect on the organisation's work; i.e., in lost business during the denial of service, and the cost of procuring and converting to other computing alternatives.

Protection against denial of service usually relies on procedures in the computer centre itself. Denial of service in the case of single terminals or distributed data is less likely to incur a large cost.

Sensitive Data

A typical computer owned by a commercial organisation will contain sensitive data of some sort, e.g., a company's proprietary process, trade secrets and customer lists. Illegal access to personnel data held on a computer could help a competitor lure away key people. An organisation may be able to quantify the loss owing to disclosure of sensitive data by estimating lost profits or lost revenues.

5.1.3 Justifying Security

The justification for security will depend on many factors (including existing security arrangements and non-monetary factors), and the type and level of security adopted can be assessed by using risk analysis procedures. Multiplying the prospective cost to an organisation of a security breach by its probability of occurring produces a useful estimate of risk. Although increased security decreases the probability of a breach, there is a diminishing return with each additional security step.

Diminishing Returns

An initial investment in security is usually justified by the significant reduction in risk that it produces. At a certain level of investment, increased security may not reduce the probability of loss enough to justify the additional investment.

Existing Security

Most organisations already have a security policy in place for non-computer-related assets. For example, they may restrict physical access to the workplace and require identification badges. Large multi-user systems are often already located within a dedicated environment, such as a laboratory, and it is often inexpensive to establish tighter physical security over such an area.

Non-Monetary Factors

There are factors other than quantifiable losses that may prompt an organisation to impose or tighten a security policy:

- The viability of the organisation may be so threatened by certain types of security breach that management must elect to guard against them even though they may be unlikely.
- Contracts with customers or with suppliers of software for the computer may specify a degree of security that the organisation must take in order to protect the confidentiality of data.
- Duty of care. If a computer contains personnel data, the employees may have a legal right to compel the company to take certain precautions to protect their privacy.
- Goodwill and trust. Organisations in sensitive fields, such as banking, would suffer a loss of credibility if a publicised security breach occurred.

Other Approaches

In rare cases where certain security procedures are too costly, not certain to be effective, or cannot be imposed for political reasons, management may take a different approach, such as by insurance to compensate for any loss or by arranging alternative computing facilities should a breach occur. However, these approaches are less attractive than providing security because they are reactive, while security is preventive.

5.1.4 Threats to Security

Threats to the availability, integrity and confidentiality of the system may arise from any of the following sources.

Inadvertent Breach

Security measures should be such that random typing mistakes or other common mistakes do not result in denial of service or irretrievable destruction of data.

Impatient Employees

A security policy must be mandatory enough and/or popular enough that all employees will obey it even though they may have to spend extra time doing their jobs.

Improper Employee Use

A security policy must effectively isolate sensitive data from non-authorised access. It must, for example, be able to prevent damage from being inflicted by a disgruntled employee or ex-employee.

Outside Penetration

Examine the external attachments to the computer (such as telephone and network connections) and ensure that they will adequately prevent penetration.

A security policy must be constructed so that it caters for the variety of motives for penetration. Penetration by ''hackers'' may be motivated by curiosity or intellectual challenge. Other motives include:

- *specific disclosure* Theft of data, for example, by an unscrupulous competitor, who is likely to be more determined than a hacker, but whose goals are more limited.
- *personal profit* Electronic forgery, or theft or misuse of computer services.
- *sabotage* Denial of service, perhaps by an ex-employee or terrorist.

Other Threats

A proposed physical security must guard against natural calamity, such as fire, flood or earthquake. It must protect against armed attack and provide for alternative plans in the case of disruption of routine that might occur in the case of war or civil disorder.

Transition to a Secure System

Section 6.2, **Transition to a Secure System** provides administrative procedures that review the security of an existing X/Open-compliant system.

5.2 ONGOING TASKS

The goal of security is to provide assurances of continuing protection. Such a goal requires a continuous process involving planning, communication, education and personal attitudes.

5.2.1 Planning

The security policy you devise should be in written form, and be reviewed and approved by sufficiently senior management to ensure compliance by all users. It should make provisions for each of the seven forms of security listed in **Section 1.3**, **Forms of Security**. The organisation should determine that the proposed policy protects against the threats to the system listed in **Section 5.1.3**, **Justifying Security**. To make security awareness a continuous process, the policy must operate in such a way that it can be revised as conditions change.

In this way, the organisation will effect security by design, not as a result of reactive efforts on your part.

A prerequisite for security is that the computer systems should be clearly documented. Where many machines are closely related or are operated by common staff, a uniform structure is important and takes precedence over special requests from users.

Job Descriptions

You may find it useful to include security considerations in job descriptions. Defining security duties in an employee's job description can give the employee greater motivation and make it automatic for new employees to learn the organisation's expectations for security.

Chains of Command

Chains of command are planning aids that ensure security duties are carried out even if key persons are unavailable. Overdependence on a single employee is not advisable, and therefore every important security function should have a responsible alternative person.

5.2.2 Communication

Much of your job depends on receiving information from other departments. (Much also depends on giving information; see **Section 5.2.3**, **Education**.)

New-User Information

You must define a procedure for obtaining timely information from site security, personnel, purchasing or project management on new employees and contractors who will need to use the computer.

You must define how to decide each new user's needs for computer access and special authorisations, such as physical access to the computer room. You or your operations staff interpret these requirements, assign user names, assign users to groups and set appropriate authorisations.

The process you define must respond to changes in the organisation. For example, users may need to change groups as their project assignments change. An employee writing a software package for other users may require special authorisations to install and test the package.

The process should provide for periodic re-evaluation of the authorisations that are granted to certain users. For example, access to the computer room is an attribute that can be taken away as well as given (e.g., when an employee's role changes).

The process must provide for automatic notification when an employee or consultant leaves the organisation, and comparable input from project management when a computer user ceases to work on the project concerned. This input should automatically cause a computer security officer to remove authorisation for that person. Such action is crucial because the reason the person leaves the organisation may also provide the motive for breaching security.

Site Security

You must establish regular lines of communication with the site security department, since your security policy must mesh with, and take advantage of, the organisation's existing security policy for non-computer assets.

Site security is usually responsible for physical security, which often includes emanation security (see **Section 1.3.2**, **Emanation Security**). In writing the security plan and in assuring ongoing security, you need to know what level of physical security is already present. You need to inform site security if additional security is needed to physically protect the computer room.

Building Engineering

Building engineering, for the computer room or laboratory, is an aspect of physical security for which the site security department may not be responsible. You may instead have to work with the site engineers. To assure availability, one should protect against incidents caused by natural or man-made disasters.

Remote Connections

If you have communications lines to other computing centres, establish contact with the administrator at these centres. Regular meetings with these administrators may be useful to exchange information on security issues. This may help to avoid errors which have already been encountered.

Should one of the centres be the subject of an attack, it should warn the others as soon as possible. For this purpose, you must establish a route that allows rapid notification of the administrator and a means by which the authenticity of the notification can be verified. This is likely to involve a ''callback''. Where such notifications are authentic, you may wish to close down the service for a certain time.

5.2.3 Education

Management

As the primary advocate of security, you must often educate management on the true value of the computer system and the need for a security policy. Preservation of integrity is a potent selling point for establishing a security policy, but you may have to overcome the resistance to some effects created by the security imposed. For example, management may value an informal work environment, or, once security is imposed, managers may find they are impeded by it, since a management role in the organisation does not necessarily justify security authorisations on the computer.

You must make management aware that occasionally you may require their support to establish cooperation from all classes of computer user.

Operations Staff

A security policy must include a process by which new employees learn the organisation's security policy. Since special authorisations are a function of an employee's role, not seniority, new employees in certain jobs may acquire authorisations shortly after they are hired. This should not occur until they have received suitable training on the security implications of their authorisations.

A new operator may be asked to take an action that has security implications. Even before the nature of that action is learnt, the operator should know the security policy well enough to know whether or not to seek administrative approval for the action.

System Programmers

The security policy puts constraints on the design, coding and review of system programs (see **Chapter 4**, **Security for Programmers**), and you must define a way for these constraints to be routinely conveyed to programmers. Teaching materials will be more effective if they include examples of both proper and improper programs.

The educational effort should be directed at all phases of program development, because it is easier to design a program with security in mind than it is to add security as an afterthought.

To help overcome resistance that might be met, the educational effort should emphasise the benefit of the continuing availability of the computer and the preservation of the programmer's work. Some programmers view security as an issue that is not germane to their assigned task, while some prefer to concentrate on technical issues, perceiving security as a personnel or political issue.

Users

All users must take action to make security work (see **Chapter 3**, **Security for Users**). While or before learning their task on the computer, new users should learn routine security procedures.

You can help new users by providing shell scripts, procedures to protect files and warnings about executing programs.

Updates

Security policy may change from time to time, as the organisation receives new hardware and software, as the use of the system changes and as dictated by actual breaches.

The education plan must include a way of conveying policy changes to each of the groups listed in this section.

5.2.4 Attitudes

A security policy imposes short-term costs on all computer users. (In this context, costs refer to, for example, the loss of freedom incurred by restricted access or the extra work involved in making something secure.) Although a well-devised security policy is worth while, its benefit is intangible and long-term, and not all users will perceive its worthiness at all times.

To foster universal compliance with the security policy, you should consider the effects of your actions, including subliminal or motivational effects. In particular, the following points should be borne in mind.

Avoid Favouritism

You seek to impose a security discipline on all users. Users will accept the policy most readily if they perceive that the policy treats all similarly situated users equally. Over time, the organisation may come to trust different employees to different degrees. But you will give widest effect to the security policy by avoiding the appearance that application of the policy is based on a level of trust. This may avoid the situation whereby users feel that being asked to obey certain rules implies a lack of trust.

It motivates by example for employees whom the organisation clearly trusts, notably the administrator, to scrupulously obey the rules.

It harms motivation if you seem hypocritical or do not elect to pay the costs that the policy imposes on others. For example, an administrator who gave relatives a tour of the facility, without the clearances that other employees would require, would produce resentment.

Use Courtesy

An employee's opinion of the administrator or operations staff, in the extreme case, can become a motive for disobeying security policy or trying to penetrate the system. It is therefore important that you apply the security policy with courtesy. In particular, do not make accusations that you cannot substantiate.

Avoid Boasting

Conspicuously advertising the level of security you have achieved may in itself motivate someone to try to penetrate the system.

5.3 SECURITY BREACHES

If a breach occurs, restoring security has the highest priority and demands immediate action.

5.3.1 Identifying the Breach

You may discover a potential security problem in the following ways:

- Review of a log file shows unusual activity on the computer, indicating possible unauthorised use.
- A user reports a loss of data or denial of service. A particular example is any case of a complete system crash.
- An event outside the computer environment occurs suggesting that unauthorised disclosure of computer data has occurred.

5.3.2 Determining the Cause

Restoring lost data will not close the security breach. You must identify the cause of the loss and take steps to prevent its recurrence.

The first step is to examine the audit logs that recorded events around the time of the breach. They may show unusual user behaviour or identify unauthorised use of special privileges. Information from system users and from ‘‘sign-in’’ sheets may indicate improper use by imposters.

Examine other sources of information, such as handwritten logs and hardware field-service records. You may find a correlation here that could identify the cause of the problem.

5.3.3 Repairing the Breach

If you can prove that the failure affected only a portion of the system, such as a single file system, you can limit the amount of restoration needed. Otherwise, you must work to restore the integrity of the entire file system.

If you cannot trace the cause of the problem, restoring integrity involves an analysis of every sensitive data file and restoration of every security-related program in the affected portion, either by recompiling or rebuilding programs, or copying known unbreached versions from distribution media or from secure backup media. (Observe the cautions in **Section 3.5.5**, **Backup and Restore** about the inherent security problems of backup media.) You should also restore the operating system code from a known unbreached version, and use diagnostic programs supplied by the vendor in order to eliminate the machine as a possible cause of the problem.

If the organisation has achieved true security even in the absence of problems, you can back up files from a known date and assure users that the problem will not recur. Otherwise, you may be replacing corrupted software with equally corrupted software.

5.3.4 User Cooperation

User cooperation is typically needed to identify the cause of the breach and replace any lost data. You must notify users of the impact the breach will have on them. If you

cannot identify the cause, you must inform users of the possibility of continuing problems, and of the timetable for restoring the system.

In any malicious or unexplained breach, require all users to select new passwords as though they were new users. If your system has password aging, declare all passwords to be aged, so that the system itself requires each user to select a new password.

5.3.5 Responsibility

If the breach was willful and caused a loss, you will want to hold the penetrator responsible. In the case of an external penetration, such as by a competitor, the organisation may sue, and you will have to participate to enable the recovery of damages. Even in cases where the organisation cannot win restitution for its losses, you will want to determine who is responsible as a means of strengthening the security policy.

To allow your company to sue a penetrator, you must be prepared to give proper evidence in court. The necessary actions, such as collection and storage of data for such a purpose, should be part of the written security policy.

Chapter 6

Administrative Procedures

6.1 PRIVILEGES

Many administrative procedures, if misused, could deny service to users, or corrupt or disclose files. On a secure system, general users cannot do these things. The administrator cannot do them either without first obtaining special authorisations. The principle of least privilege dictates that a process should acquire only those privileges needed, and only for as long as needed.

6.1.1 The Super-User

As a super-user you obtain appropriate authorisations by logging-in or switching to another account with authorisations. **Section 2.1**, **Users** describes this mechanism and explains how it can be implemented using **/etc/passwd** to hold user and authentication data.

A user must type a correct super-user password before acquiring appropriate authorisations. You should select and protect super-user passwords in the same way as specified in **Section 3.2**, **Password** for general passwords. Disclosure of the super-user password would have system-wide impact on security. It is necessary to change this password frequently.

The principle of least privilege dictates that you use privilege only when necessary and only for as long as necessary. The principle of personal accountability dictates that, each time you use the super-user account, you do it in a way that identifies you personally. There are several techniques for letting authorised users acquire super-user authorisations:

- Have a single super-user account, with user name ‘‘root’’. You should require users to use a utility such as */bin/su*, not *login*, to switch to the ‘‘root’’ account. Users establish the super-user shell as a subprocess of their normal environment, and have already used *login* to record their identity. This method produces better logging of the use of privileges if your */bin/su* maintains a **sulog** log file. It also makes it easy for users to revert to their login environment for routine procedures. Finally, there is a single password giving access to super-user authorisations. In the case of a security breach, you can quickly raise the system’s defences by changing this password.

  A variant of this scheme specifies two administrator accounts. One of the accounts is called ‘‘root’’, has a non-zero UID and cannot be used for login (e.g., it has an impossible value of password). It is defined only for compatibility purposes. The other account has a name known only by the administrator, has a UID of zero and can be used to login. This increases the effort required by a penetrator.

- You can create several super-user accounts in the user database, giving a specific individual responsibility for each one. Users should also have regular accounts for actions that do not require privileges. If you experience problems with disclosure of super-user passwords, this approach lets you trace the problem.

- Some advanced versions of */bin/su* require entry of the correct super-user password plus additional authentication.
- If possible, try to separate administrative tasks and associate them with different accounts. Using set-user-ID programs tailored to the needs of operators, authentication administrators, etc., avoids the need to give super-user privileges to the account.

Guidelines for Becoming Super-User

- Plan your actions before becoming super-user. This helps ensure that when you complete these actions, you will remember to revert to your login shell. Also, in case of problems, you will know exactly what you planned and what you did.
- Understand what you are about to do. If you are about to become super-user at the request of another user, you should understand the nature and implications of the request. If it seems unusual or appears to contradict these guidelines, obtain independent confirmation. For example, operators should seek approval from superiors. Managers should read reference documentation in order to understand the nature of the request.
- Do not delegate any task that requires super-user power: do it yourself.
- Try to use a terminal that is dedicated to administrators and physically and visually protected from other users. Consider any unusual response during your login to be a security breach. Find out what happened and why before you continue logging in or take any action as super-user. Note especially the previous user of the terminal.
- Always type the full pathname for commands like */bin/su*. Do not just type *su*, even if it works. Depending on the integrity of your search path and the directories it points to (see **Section 3.4.2**, **Search Path**), you could be tricked into running a Trojan horse program planted by a penetrator (see **Section 1.1.6**, **Types of Penetration**). It would typically capture the *su* password for the penetrator's use.

Guidelines for Super-User Sessions

The intent of these guidelines is to make it impossible for you to execute a penetrator's program while you have super-user power.

- Use the ''-'' option of *su*. That is, type:

  ```
  $ /bin/su -
  ```

  This obtains the normal environment expected by the super-user account. **Section 3.4**, **Secure Environment** explains the importance of a secure environment. For example, it obtains the correct working directory and search path.
- Use full pathnames to specify all programs you invoke as super-user.
- As super-user, never execute a program whose identity is in doubt, whose effect is unknown to you, that has not been tested and reviewed or that resides outside a system directory.

6.1.2 Single-User Mode

Many X/Open-compliant systems allow single-user mode. The system may enter single-user mode before time-sharing begins. The super-user can invoke single-user mode in order to end time-sharing.

Single-user mode lets you perform actions such as backups, software installation and file reorganisation, without the possibility of interference from or to sessions of general users. These operations do not necessarily require super-user authorisations. When you operate in single-user mode, you typically have super-user authorisation. When doing something that does not require this authorisation, such as a backup, you should invoke */bin/su* as described in **Section 6.1.4**, **Switch to Another User**. In this situation */bin/su* obtains a *lower* authorisation (so that any mistakes have the minimum impact on the system).

In single-user mode, the shell appears at the console (the terminal usually represented by the special file **/dev/console**).

On a system that restarts in single-user mode, a penetrator with physical access to the console and to the computer can bypass all software security to force a restart.

Daemons typically call *sync*( ) periodically to write changed data from memory to disk.

In single-user mode, daemons typically do not run, and so you should synchronise the filestore manually after changing a disk file, to ensure that your change is applied to the file system[1].

6.1.3 Pseudo-Users

For many tasks, all the user needs is ownership of specific resources. You typically define a variety of pseudo-user accounts for this purpose, rather than super-user power. A pseudo-user is an account in the user database that does not correspond to a human user of the computer. A pseudo-user has a password, a home directory and a login shell. Like all other accounts, a particular user should have ultimate responsibility for actions taken under a pseudo-user name. Pseudo-user accounts should not be shared.

A typical X/Open-compliant system has the pseudo-user account ‘‘root’’, discussed in **Section 6.1.1**, **The Super-User**. The system may have other pseudo-user accounts, without super-user power, such as the following:

acct The account that owns the accounting logs.

bin The account that owns most of the programs and system directories.

lp The account that maintains the printer spooling system. Other pseudo-user accounts may exist for other spooled devices or systems.

mail The account used by the *mail* administrator. It is the owner of the *mail* utilities and associated data files.

uucp The account that handles the *uucp* subsystem. It needs help from ‘‘root’’ to

1. On many systems this can be done by calling the *sync* utility, or, from the program level, by calling the *sync*() function.

establish login accounts for remote computers.

As with the ‘‘root’’ account, divide any pseudo-user account into however many accounts you need so that each person has a separate account. This permits accountability. If you do so, you must set up groups of pseudo-users, as described in **Section 6.3.5**, **Adding Groups**.

For example, with separate accounts, pseudo-users ‘‘uucp1’’ and ‘‘uucp2’’ both have access (through group permissions) to the *uucp* subsystem. Typical X/Open-compliant systems have the following groups:

admin	The group for most system files and directories. Any object that does not need different group and other permissions can be in this group.
lp	The group for line printer administrators.
mail	The group for mail administrators. Mail administrators can maintain the **/usr/mail** spool directory.
uucp	The group for *uucp* administrators.

6.1.4 Switch to Another User

Section 6.1.1, **The Super-User** describes how */bin/su* can be used to become the super-user. Most implementations of */bin/su* allow an argument to be specified, so you can assume the identity of another user.

The form of the command is:

```
$ /bin/su - new_login_name
```

The ‘‘**-**’’ option ensures that you will use the environment that the specified user would have if logging in normally. Omitting ‘‘**-**’’ retains your current environment. This can conflict with other attributes of the specified user. It could result in your executing bogus code planted by a penetrator.

The */bin/su* utility requires you to type the correct password for the desired user (unless you were the super-user).

You should use this form of */bin/su* for two purposes:

- To reduce your authorisation when operating in single-user mode (see **Section 6.1.2**, **Single-User Mode**) if you do not need the super-user power that comes with single-user mode.
- To make a transition into a pseudo-user account (see **Section 6.1.3**, **Pseudo-Users**).

Do not use */bin/su* to assume the identity of a general user. Although you might want to do this in order to change the ownership, attributes or permissions of an object that the user owns, there are two adequate alternatives: (1) have the actual user assist in the operation, or (2) become the super-user. These alternatives make it unnecessary for any administrator or operator to learn the password of a general user.

6.2 TRANSITION TO A SECURE SYSTEM

This section describes the aspects that should be reviewed when making the transition to a secure system.

6.2.1 Users

Your review must determine who is authorised to use the computer and for what purpose. You must change the user database to reflect the current set of users. You will typically find several lines that correspond to former employees or to modes of operation that are no longer relevant (for example, obsolete pseudo-user accounts).

On implementations that map user and authentication data into a file such as **/etc/passwd,** no password entry should be null. You can quickly detect violations of this rule by typing:

```
$ grep '^[^:]*::' /etc/passwd
```

If you are using a system that permits password aging, then you should enable it.

6.2.2 Groups

Section 2.2, **Groups** describes the group database and a common implementation using **/etc/group**. Administrators often fail to keep this file current. This file must contain valid data, based on considerations such as project team assignments that require users to share files.

Define a separate group, containing only one user, for each user who does not need to share files.[2]

6.2.3 Accounts

Reviewing an account includes examining the user's directories (see below) and the user's initial environment given in the **.profile** file. Suggestions for the contents of **.profile** are given in **Section 3.4.1**, **Profile Files**. Users prefer total control of their files, but **.profile** is one case where you share responsibility and can justify making changes to a user's file if its contents threaten security. You should also assign appropriate permissions to each user's home directory. Inform the file's owner before making any such changes.

6.2.4 Directories

Examine all user hierarchies for files owned by that user and containing the set-user-ID or set-group-ID attribute. Understand the purpose of each such file. If these files are being used to effect sharing of data, create adequate groups and ask users to apply group permissions to the files they want to share.

---

2. This makes it easy to implement group file sharing at a later date if desired. If everyone is put in the same group, dividing users into groups later might require that you apply *chgrp* or *chmod* to existing files to prevent group access.

A general user's directories must never contain set-user-ID programs with some other user's UID, or especially the super-user UID. These programs can give a general user great power. The existence of these programs in these locations undermines security. If the user has a reason to have such authorisation, the user should obtain authorisations in a more conventional and easily monitored way. A general user must not have write access to a directory in the system default search path (e.g., as set by **/etc/profile**).

6.2.5 Pseudo-Users

Protect certain security-related programs and data files by using pseudo-users. Create a user account in the user database that does not correspond to any real user. Make this pseudo-user the owner of a specific class of security-related objects. Then you can give a user access to that class without having to give out unlimited authorisation.

6.2.6 Program Binaries

Review the program code of all sensitive programs. In particular, review programs that have the set-user-ID, set-group-ID or privilege attributes, making sure they follow the precautions given in **Section 4.4**, **Privileged Programs**.

If you must install third-party software with privileges and you have only the executable file, search it for references to pathnames. If your system has the *strings* utility, which searches for printable strings in an executable file, you can type:

```
$ strings - executable_file | grep /
```

6.2.7 Other Authorisation Files

If any programs have authorisation files that describe legitimate users of the program, review them and remove obsolete information. Notably, review the following:

- the **at.allow** and **at.deny** files
- the **cron.allow** and **cron.deny** files
- the appropriate *uucp* authorisation files (see **Section 6.6**, **Communication**)

6.2.8 Auditing Tools

Many programs can be directed to produce log files listing all their uses. This logging should be enabled. Create any necessary directories and assign correct permissions.

If the program is privileged, it will be able to write the log regardless of permissions. You may want to deny search permission to log directories so that ordinary users cannot easily determine which activities are logged.

6.3 ADMINISTERING USERS

The administration of users is a basic part of the job of administrator. Before a new user can use a secure computer, you must establish a user account. You must follow certain rules when setting up or changing an account to ensure isolation. Use caution when you edit account data. The consequences of errors can range from user inconvenience to a system-wide security breach.

Most X/Open-compliant systems provide an administrative package that automates tasks such as adding new users. Use of such a package reduces error. In addition, these packages typically recover from any errors without partially adding an account or failing to complete some other operation, which could leave the system in a non-secure state.

6.3.1 Adding Users

To establish a user account, you must edit the user database, create a home directory and place certain files there. The automated administrative package typically carries out these tasks.

Adding the Account

If you use an automated package, you must type certain items to describe the new user. If you do not use such a package, you must edit the user database manually. If your user database is implemented using **/etc/passwd**, enter a new line in the file format described in **Section 2.1**, **Users**. It is important that only one person edit the user database at any one time, to ensure the preservation of all changes to this file.

Login Name

Use a separate login name and a separate UID number for each user. This ensures individual accountability.

Do not choose login names beginning with numbers. Utilities like *chown* treat such names as UIDs instead. The *ls -l* command for each directory in the system should not reveal numbers in the field for login names.

Avoid login names that a penetrator would be likely to try, such as ‘‘guest’’, ‘‘system’’, ‘‘admin’’, ‘‘unix’’ or the name of the system itself or of the organisation. (There is more on guest accounts at the end of this section.) Do not choose long login names. If your system imposes a limit on the length of login names, it may ignore part of long login names, and treat two long login names as identical.

Password

The automated package typically asks you for a password for the new user. If you are administering a system that implements user and authentication data in **/etc/passwd**, type an invalid password, such as ‘‘*DISABLED*’’.

Then, after you end your edit, run a command like *passwd* to select a valid password for the new user.[3]

If your system has password aging, indicate that the new user's password is aged. This directs system access utilities such as *login* to force the user to choose a new password the first time the user logs in.[4]

If your system does not have password aging, it is important that you choose a password for the new account. Typically, leaving the password field blank or specifying a null password means that utilities like *login* will not ask for a password at all. This state is not secure. Some systems allow a user account without a password but prompt the user for a mandatory password on his first login. Do not use this feature. Select a random password for the user and privately inform him of the value. If your system does not support password aging, note the value of the encrypted password and check after some days that the user has selected a new password.

User ID — Add a unique user ID for the user. The automated package may select one for you. If you select a value manually, avoid reusing a previously used but now disabled user ID.

Group ID — Add the user to an existing group, or create a new group for the user (see **Section 6.3.5**, **Adding Groups**).

Miscellaneous Field — On implementations that support **/etc/passwd**, use the fifth field on the line to include the user's full name, as well as the user's number within the organisation or some other unique identifier. But do not put confidential information in this file, since everyone can read **/etc/passwd**. Also ensure that users are aware of this usage and alert them not to pick passwords that can be guessed using information in this field.

Home Directory — Creating a home directory is a separate step in adding a user. It is described in detail later in this section. The name of this directory must appear in the user's entry in the user database.

Initial Program — Choose the initial program that will run when the user successfully logs in. This is typically */bin/sh*, */bin/csh* or */bin/ksh*. (If you do not specify a shell, the user gets */bin/sh*.) Some systems provide a menu-oriented interface like **/bin/vsh** for novice users.

---

3. The *passwd* command is not defined in the **X/Open Portability Guide**, thus its availability and usage may vary from one system to another.

4. Set the aging interval so that all users must change passwords periodically. Frequent changing of passwords limits the access of penetrators, even if they manage to guess or steal a password. Furthermore, the knowledge that passwords are valid only for a limited time may discourage penetrators from trying this type of security breach. Some systems have techniques that keep users from changing their passwords back to a password they have used recently.

Some systems provide a restricted shell, typically called *rsh* (see *sh*). This shell takes various steps to limit the user's scope to the home directory. But a user with access to privileged programs, or to programs with escapes to other shells, can easily evade these restrictions. If the initial program is neither *sh* nor *ksh*, bear in mind that **/etc/profile** will not be executed. Hence, any security measures contained in there will not be effective for such users. Some systems allow the user to change his shell with a command. You might decide to disable this command.

When you give a user an account for the sole purpose of running an application, you may specify a program within that application as the user's shell. If that program is written correctly, this action strictly limits the user's options on the computer.

On some systems, information in **/etc/passwd** can restrict the user to a protected subsystem (see **Section 6.5.5**, **Protected Subsystems**).

Guest Accounts

Guest accounts are designed for temporary use by persons outside the organisation and each guest account should not be used by more than one person. Login names such as ''guest'' should not be used. A guest account that is not specific to a particular guest undermines personal accountability for use of the computer. The security plan should provide a way for the appropriate department, such as the sales group, to notify you when guest accounts are needed. The notification should say how many guest accounts are needed, for whom each one is required and for how long each is required. The designated person should be the only user of a guest account. Security policy should provide for automatic removal of this account at the proper time.

Home Directory

Give each user a separate home directory. This promotes individual accountability, since users are generally responsible for objects in their own home hierarchies. File sharing should always be an explicit decision; users should not share files by default.

Place the home directory in the same file system as other users in the same group(s) and away from system files. Directories superior to the user's home directory in the hierarchy should not have write permission, but must have search permission.

The home directory should have the permissions:

`drwx------` (700)

(The user has the power to widen access to the home directory.) The owner of the directory should be the user. The group of the directory should be the user's login group specified in the user database.

Environment

Most administrators place some files in the user's home directory. They typically contain useful standard settings and information on site policies. This technique ensures

universal knowledge of and compliance with security policy.

The shell runs the **.profile** script when a user logs in. You should put a suitable **.profile** in each user account. This file should start each user session with secure operating settings, even though the user has the power to change them. **Section 3.4.1**, **Profile Files** describes typical contents of **.profile**.

You can help users with previous UNIX experience understand your installation by adding to **.profile** some lines that show the values of commonly referenced variables. For example:

```
echo The terminal type is $TERM
echo Your user name is $LOGNAME
echo The path is $PATH
echo The umask setting is ‘umask‘
echo The maximum file size allowed is ‘ulimit‘
```

Do not include in **.profile** an automatic command to enter the *mail* utility. This requires a user's attention immediately after logging on and may confuse new users unfamiliar with your mail system. However, you may elect to send the user mail as you authorise the new account. The mail, perhaps a copy of a standard file, typically introduces the system and outlines policies and procedures. It may contain references to any on-line instructions.

Putting commands in a user's **.profile** gives the user a template for customising the options to the user's own needs. It also informs the user about security policy. Putting commands in a user's **.profile** does not ensure security or restrict the user, since typical users should always be able to edit their own **.profile**. Settings that are obligatory for all users should instead go in **/etc/profile**.

The file **/etc/profile**, if it exists, contains a shell script that is run by the *sh* utility when the user first logs in. (Other shells may not have a comparable script.) You can place site-specific setups here instead of in each user's **.profile**. Commands in **/etc/profile** are immune to user changes. In addition, it is easier to maintain a single global profile file as your policy changes.

For example, a *ulimit* command in **/etc/profile** cannot be removed by an affected user. The shell accepts subsequent *ulimit* commands, but only to further reduce the legal maximum file size, not increase it.

Although **/etc/profile** affects all users whose login shell is *sh*, you can use the shell **case** statement on the output from *logname* to apply different settings to different users. Most often, you want different users to have different *ulimit* settings.

If you use **/etc/profile**, you should always deny write permission to general users. You should further protect **/etc/profile** by using *trap* (and having a *trap* as the first action within the outer **trap**) to intercept signals sent to the shell from the same user at another terminal. For example:

```
trap "trap ’’ 0; exit 1" 0 1 2 3 13 15
```

Some systems tell the user when and at what terminal they last used the system. This lets users detect unauthorised uses of their user name. If your system does not do this, you can devise a substitute by adding to **/etc/profile** text comparable with the following:

```
# User auditing script
#
AUDITSIZE=20
LOGFILE=$HOME/.lastlogin
if test -r $LOGFILE
then
  echo "Last login:       ‘tail -1 $LOGFILE‘"
  datum=‘ls -lc $LOGFILE | tr -d "-"‘
  set $datum
  if test $7 -lt 10
  then
    echo "Audit record written: $6  $7 $8\c"
    echo "             USER: $3"
  else
    echo "Audit record written: $6 $7 $8\c"
    echo "             USER: $3"
  fi
  AUDITLENGTH=‘wc -l < $LOGFILE
  if test $AUDITLENGTH -ge $AUDITSIZE
  then
    AUDITLENGTH=‘expr $AUDITSIZE / 2‘
    tail -$AUDITLENGTH $LOGFILE >${LOGFILE}2
    chmod u+rw,go-rw,a-xs  $LOGFILE
    mv ${LOGFILE}2 $LOGFILE
  fi

elif test -f $LOGFILE
then
  echo "$LOGFILE is improperly protected."
  exit 1
else
  echo "$LOGNAME has never logged in before."
  > $LOGFILE
fi
chmod u+rw,go-rw,a-xs  $LOGFILE
echo "‘date‘ USER: $LOGNAME TTY: ‘tty‘" >>$LOGFILE
chmod u-w $LOGFILE
#
# End of user auditing script
```

Now the file **.lastlogin** provides a history of logins by that user. (This file is placed in a directory with the user's login name, as given by *LOGNAME*, in case several users have the same *HOME* directory.) Each user that executes this script will see output in this form:

```
Last login:        Mon Aug 08 08:08:08 MEZ 1988   USER: Mary TTY: tty08
Audit record written:  Aug 08 08:08               USER: Mary
```

The output shows both the time of the user's last login and the time at which **.lastlogin** was last modified. If a penetrator logs on as the user, and then modifies the user's **.lastlogin** file to remove the record of the penetration, the two dates/times displayed by

the script will no longer match. Direct all users to study the output of this script each time they log in. Users should immediately log out and report a possible penetration if: (1) the dates, times or user names on the two lines are not identical; (2) the user name on either line is not correct; or (3) the dates / times are not those of the user's last login.

6.3.2 Disabling an Account

Disabling an account means ensuring that there are no active users. This is typically done before removing an account and before moving a user's files to another place in the file system. You disable the account if time-sharing must continue during your operation, to be certain your actions do not disturb other users. You need not disable the account if you instead put the entire system into single-user mode.

To disable an account, follow this sequence:

1. Invalidate the user's password. For example, edit **/etc/passwd**, adding to the start of the encrypted password field a standard legend, such as *DISABLED*.[5]
2. Invoke the *who* utility to see if the user is logged in. If so, become the super-user and use *write* or other form of communication to ask the user to log out.
3. Verify that the user has no active processes, by typing:

   ```
   $ ps -fu login_name
   ```

   If *ps* shows current jobs, kill them or wait until they terminate before operating on the user's account.
4. Unmount all mounted file systems appearing in the user's hierarchy. (You may have to take the system into single-user mode to close files, unmount systems and eliminate current directories in the user file system.)

Re-enabling an account is done by editing the user's password again, and deleting the legend *DISABLED*.

6.3.3 Removing an Account

You should remove an account when a user leaves the organisation or ceases to require access to the computer. Removing inactive accounts reduces ways to penetrate the system and makes it more likely that operators will understand the use of, and need for, all accounts that remain.

Before you remove an account, disable it, as described in **Section 6.3.2**, **Disabling an Account**, or put the system into single-user mode. In this way, your actions will not disturb other users. Then do the following:

1. Make backup copies of the user's data, or ensure that a current backup has covered the user's data.

---

5. Some installations instead disable an account by specifying a program without user input, such as **/bin/date**, as the login shell. This keeps the user from doing anything meaningful with the computer. But changing the password field has the advantage of giving a penetrator a more ambiguous response.

2. Determine whether any of the user's data might be needed by other users. This could include any objects in the user's hierarchy as well as mail messages. Copy or provide for the retention of these data. This step may be the user's responsibility. Make sure that the files will belong to another valid user.

3. Notify other users, especially members of the same group, that you intend to remove the account.

   Users who share files you are about to remove may need to make copies of them. Users may also have to adjust absolute pathnames and environment variables. The correct amount of advance notice to give depends on how interdependent users are and on their workload.

4. The user (or you, after using a command like */bin/su* to gain access to the user's account) should remove the user's *crontab* and *at* jobs:

   ```
   $ crontab -r
   $ at -l
   $ at -r job_number
   ```

   Although unlikely, the user may still have print jobs in the printer queues. If so, delete them.

5. Remove all the files and other objects in the user's hierarchy, using the recursive form of *rm*:

   ```
   # rm -rf home_directory
   ```

6. Become the super-user, and locate any objects owned by the user but residing in other users' hierarchies:

   ```
   # find / -user login_name -exec rm -rf {} \;
   ```

   Remove them or change their ownership with *chown* to coincide with the directory in which they reside.

7. Delete references to the user from the administrative files:

   - **user database** - remove the entire entry that defined the user. Use the same precautions as you do when adding users (see **Section 6.3.1**, **Adding Users**).
   - **group database** - remove all references to the user. If this leaves no authorised members of one or more groups, delete those groups.
   - **cron.allow** and **cron.deny**.
   - **at.allow** and **at.deny**.
   - User lists used by *mail*, and mailing lists.

There should now be no active references to the user in the system. (References may reappear after you restore backup files from disk or tape; see the note at the end of **Section 6.3.6**, **Removing Groups**.)

If the *ls -l* command ever prints numbers in place of the owner name, it means you deleted a user without accounting for the user's objects stored outside the user's hierarchy.

Delete that object, add the user again or change the object's ownership with *chown*.

6.3.4 Moving User Hierarchies

The typical reason for moving a user's directories is that your strategy for allocating disk space has changed. For example, a group may begin to fill its assigned disk, so you would need to move some group members to another disk.[6]

As the super-user, you can see how much space a user's hierarchy takes by typing:

```
# du old_home_directory
```

You can see how much space is available in the new file system by typing:

```
# df new_file_system
```

If these commands determine that the move is physically impossible, make other plans. Be sure the new file system has enough room to spare so that normal activity during the move will not exhaust it.

Before you move hierarchies, disable the account, as described in **Section 6.3.2**, **Disabling an Account**, or put the system into single-user mode. In this way, your actions will not disturb other users.

Then become the super-user and follow this sequence:

1. Record the permissions that *ls -l* shows for the home directory. Change its permissions to

   d - - - - - - - - - (000)

   denying everyone access to the directory.

2. Copy the old hierarchy to its new location, preserving all modes, owners and permissions:

   ```
   # cd old_home_directory
   # find . -depth -print | cpio -pdlma new_home_directory
   ```

   If you are moving a hierarchy within the same file system, you can simply type:

   ```
   # mv old_home_directory new_home_directory
   ```

3. Verify that the new home directory has the same permissions as you previously recorded for the old home directory. If it does not, change them with *chmod*.

4. If you moved the hierarchy to a new file system using *cpio*, remove the hierarchy from the old file system by typing:

   ```
   # rm -rf old_home_directory
   ```

---

6. **Section 3.5.5**, **Backup and Restore** contains precautions for moving hierarchies between installations. On different installations, an object's attributes may not be appropriate and its UID and GID may no longer be correct. The administrator of the target system must correct them after restoring the objects from disk or tape.

5. Edit the user database and specify the new location of the home directory. If you disabled the account, re-enable it after editing with the *passwd* command, or equivalent.
6. Notify the user that the move is complete.

6.3.5 Adding Groups

Section 2.2, **Groups** defines groups and provides an example of a group database using **/etc/group**. You should add a group to this database any time a new project, class or organisation joins the user base. This process is typically automated by your administrative package. If you add a group to the group database manually, follow these rules:

- Avoid simultaneous updates to the group database.
- Give every group a unique group name and GID number.
- Do not pick group names beginning with numbers. Utilities like *chgrp* treat such names as GIDs instead.
- Preserve the relationship discussed in **Section 2.2**, **Groups** between the user database and the group database. For example, be sure each user's login group listed in the user database is a valid group. Do not use group passwords.

6.3.6 Removing Groups

A group is removed by deleting its entry from the group database. Follow the same guidelines as you do when adding groups.

Groups with no members are candidates for removal, but there may still be objects with the group's GID and other references to the group name. To find them, become the super-user (in order to search the entire file system) and type:

```
# find / -group group_name -print
```

If there are many references, you may decide not to remove the group but just to disable it. Do this by removing the list of members. For example, on a system that implements the group database in **/etc/group**, to deactivate the group defined by:

```
stockboys::800:dave,clarence
```

you replace it with:

```
stockboys::800:
```

If the *ls -l* command ever prints numbers in place of an object's group name, it means you deleted a group without accounting for objects with that GID. Delete that object, re-insert the group or change the object's group with *chgrp*.

You typically find references to a removed group on restored files. The backup media may have been written before you removed the group. It is more convenient to deal with these references after the files have been restored, rather than to inspect every backup tape every time you remove a group.

6.3.7 Auditing Users

You must be constantly aware of the types of activity the computer is being used for, the mode of use and the typical workload. In this way you are more likely to notice unusual activity and detect any breach of security.

Many X/Open-compliant systems produce logs that contain useful security information. These include **sulog**, **/etc/utmp** and the logs produced by *uucp*. System access utilities like *login* may record date, time, duration and terminal of each user login in **/etc/wtmp**.

On many systems, the *cron* utility starts jobs based on *at* or *crontab*. It may log usage in **/usr/lib/cron/log**. Some systems provide for logging of hardware faults such as disk errors. Some installations condense the information in these logs for use in billing users. You should review the raw data regularly, and **sulog** in particular. You should understand every case in which a user needs to become the super-user and be satisfied that the use of */bin/su* was warranted.

Become familiar with the ways you can use *ps*, *who*, *find* and *ls* to monitor the activities of users and of file systems.

Monitoring User Logins

You should monitor user logins using *who*. This utility typically produces one line for each user of the system; each line starts with the user name. One way to use *who* is to pipe its output through *sort* and look for duplicate user names. Depending on the format of the output from *who*, you can devise a script to extract user names, filter out other information and detect duplicate user names, perhaps with *uniq*. This script should produce no output.

If two terminals are logged in under the same user name, either someone is sharing a user name, which ought not to happen; or there is an inactive terminal or suspended process which a penetrator could come to control; or there is an imposter presently using the computer. If you detect that a user has divulged his password, lock his account and take the actions defined in your security policy for this case (e.g., inform the user and/or his manager, unlock the account only after a written request).

If there is a log file for failed login attempts, carefully analyse it.

Attitudes

Monitoring use of the computer is an intrinsic part of your job. No user should view this activity as intrusive. Users have a right to keep sensitive data from you, and a right to keep you from assuming their identity. However, you have a right to monitor the extent and form of their computer use, since your ability to detect abnormalities protects their continued access to the computer.

Post-Crash Analysis

Log files may show the state of the system at the time of a crash. For example, if session accounting was enabled, the following command shows you who was logged in at the time of the crash:

```
$ who -a /etc/wtmp
```

If your system provides a crash dump facility like */etc/crash*, you may wish to use it for further information about the state of the system and the technical cause of the crash. This may, however, require some technical expertise.

Cleaning Out Old Logs

Accumulating user data that are too detailed, or saving data on-line for too long, lessens your ability to assure users of ongoing disk space for their work. Log files that have attained a certain age should be moved to off-line storage such as tape. Do not simply remove old log files, if tape is available. Some security breaches do not become evident until long after the penetration event; maintaining old logs off-line can provide proof of a penetration.

The security policy should determine what logging data should be retained and for how long. Extrapolate from current statistics on the computer's use to determine how much storage you will need for logs.

Charges

Some organisations permit inter-departmental billing for computer use. The data processing department may assess other departments for the amount that members of those departments use scarce computing resources, such as processor time, connect time, disk space, time on shared devices and specific services. If courtesy fails, such charges may be the only motivation for users to use the computer economically. Whether such charges are necessary depends on how much the usage load affects your ability to assure users continuity of service.

6.4 MACHINE SECURITY

There are two reasons for applying security to the computer and its peripherals: (1) to effect efficient sharing of the devices, and (2) to prevent unauthorised or unstructured use of devices, which can breach overall security or deny service to other users for long periods of time.

X/Open-compliant systems provide software security over the computer's peripherals using *device special files*. A user's permission to such objects determines the user's power to use the peripherals, as described in **Section 2.4**, **Objects and Permissions**.

The rest of this section describes the physical and administrative security procedures you should take concerning the computer and its peripherals.

6.4.1 The Computer

The computer itself should be in a physically secure environment. You should strictly control access to the computer room, since it may be easy for a person with physical access to the computer to deny service regardless of their level of knowledge.

The operating system automatically shares the computer among all users. The *nice*( ) call modifies a process's priority, which controls the operating system's division of computing time among processes. You can modify a user's login shell to call *nice*( ) before invoking the real shell, to assign each user an inherent process priority. This priority applies to subprocesses created by the user. Teach users how to use *nice*( ) to reduce further the priority of their long or non-interactive jobs.

Occasionally a process malfunctions, requests large amounts of sharable resources (such as computing time, memory or subprocesses), and cannot be stopped by the user that started it. On most systems, you can use the *ps* command to identify such jobs. Confirm the situation by consulting with the user. Then invoke *kill* to remove such jobs.

Some systems provide process accounting and the *acct*() function to switch accounting on or off. You can use accounting logs to assess each user's demand on the processor.

6.4.2 The Console

The console is the terminal usually referred to by the device special file **/dev/console**. The console prints system diagnostic messages and is the single active terminal during start-up. In addition, some systems can send security-related logging messages to the console as well as to a file. This ensures that a penetrator cannot remove evidence of a penetration, regardless of the extent to which software security has been breached, without physical access to the console.

On large systems, the console may be a printing terminal located close to the computer front panel, directly wired to the computer and subject to the same access control as the computer itself. You should save the console output, especially if it includes security-related log data. In this case, supply the console with continuous-form paper. Operators can review the print-out, but instruct them not to detach and remove individual pages. Do not use the console for routine operations. This makes it harder to scan the printed log for security events. Treat the printed console output as confidential, since it could disclose procedures to a penetrator.

6.4.3 Other Terminals

Users have access to the terminals at which they log in. Most user terminals are exclusively controlled by one person. You may designate some terminals as shared. Sharing is typically governed by physical presence and moderated only by courtesy. You can review session logs produced by utilities such as *login* to determine whether some users are overusing shared terminals.

Processes gain access to terminals using device special files. Special files that represent unused terminals are typically owned by the super-user. General users should have no permission to these files. The file **/dev/tty** always refers to the user's own (login) terminal. All users should have read and write permission to this file.

The *uucp* subsystem typically uses terminal lines to communicate with other computers.

Usually there is a dedicated set of terminal lines for *uucp*. These lines are directly wired to the other computers or to telephone modems. The special files that represent these terminal lines should be owned by the pseudo-user **uucp** or belong to the **uucp** group. The user and group should have ''`rw-`'' permission to the files. General users should have no permission to these files. General users use privileged programs to gain access to remote computers.

6.4.4 Tape Drives

Device special files representing tape drives generally give ''`rw-`'' permission to all users. X/Open-compliant systems generally enforce exclusive use of a drive.

Encourage users to unmount their tapes as soon as they have finished using them. In larger systems, set aside a tape drive exclusively for backups. The special file representing this device should belong to the **backup** group and given the permissions

`---rw----` (060)

which exclude all users outside that group.

6.4.5 Other Shared Devices

Other shared devices follow the same model as tape drives. You can use permissions and privileged programs to structure the sharing of these devices. Devices such as plotters may require exclusive ownership. Devices such as optical discs, networks and external buses may not. The privileged program and the device handler ensure that general users operate the devices properly. As with tape drives, these programs may keep logs of device usage. Spooling systems are examples of programs that manage shared devices.

6.4.6 Discs

On most systems, discs, or partitions of discs, are added to the file system by using a *mount* utility. After mounting, users refer to objects on discs by specifying pathnames, and are subject to access control using permissions.

Device special files achieve raw (non-file-system) access to discs. Special files may gain access to individual characters or to individual blocks of the disk. A different special file exists for each device. Sometimes a special file exists for a single partition of a disk.

Unauthorised access to a disk special file enables access to every file on that disk or partition.

The only user that needs access to these special files is the super-user. This user also owns a tape special file, allowing transfers of data between these devices. Therefore, disk special files should be owned by the super-user and have the permissions

`rw-------` (600)

Transfers between disk and tape should be automated (typically by a shell script) to minimise error. This script may run automatically when the tape administrator logs in, preventing any unstructured use of that account.

6.4.7 Disc Space Control

You should try to assure adequate free space on each writable file system for normal usage.

A file system that runs out of space denies service to users who want to store more data on it. More importantly, if it runs out of space in the middle of an update, it can lose its integrity (see **Section 6.5.2**, **File Systems**). Furthermore, this event can crash the system, abruptly ending time-sharing and denying service to all users.

The *df* utility shows how much space is free. File systems that are small, actively used or unpredictable (such as **/tmp** file systems) should be monitored more frequently.

To assess each user's disk usage, you can use the *du* utility. For example, the following command, applied to the parent directory of the user home directories, lists grand totals of disk usage:

```
$ du -s user_parent_directory | sort -n
```

Users can be the owners of objects in other users' hierarchies.

On some systems, a user can also use *chown* to give away an object. The above technique lets you assess the contents of each user's hierarchy. It is not easy to assess disk usage by object owner, if users own objects outside their hierarchies.

You can suggest ways for users to share data files or consolidate disk usage. You can identify objects that can be moved to off-line media using *find*. For example, the following command identifies all objects that have not been read or written in the past 30 days:

```
# find / -atime +30 -print
```

Users should remove or consolidate their own files. Unilateral action by you may hamper productivity and produce resentment.

Removing old security logs and accounting logs to off-line storage is another way to free disk space. This is discussed in **Section 6.3.7**, **Auditing Users**.

On some X/Open-compliant systems, you can apply resource quotas to users. This limits the amount of disk space a user can obtain. Proper use of resource quotas can prevent crashes caused by full discs.

Memory is represented by device special files such as **/dev/mem** and **/dev/kmem**. These files are typically used only by privileged utilities such as *ps*. No ordinary user or group

should ever have any permission to them.

You must shut down time-sharing or the machine itself before making certain adjustments or maintenance of hardware or software. Certain general rules apply to all systems. The goal of these rules is to ensure that the system is totally idle before you stop it.

- Users should be given ample advance notice of the shutdown and all should log themselves out before you start final shutdown procedures. If possible on your system, disable new logins as existing users log out.
- Be sure that daemons are removed before you start final shutdown procedures.
- Look and listen to the computer to verify that it seems idle.
- Halt the computer before spinning down any disk drives. On some systems, this sequence may prevent an interruption during a disk write.

6.5 STORAGE

You must control the permissions of directories, and the permissions and contents of certain files, to keep the computer secure.[7]

6.5.1 Directories

Section 2.5, **Access Rules** describes what permissions mean for directories, and contrasts that with permissions applied to the objects in a directory.

Directories are classified in two ways. The first is based on whether or not the owner has write permission:

- Static directories have a fixed inventory of files. The files' sizes may change, but files are not added to or removed from the directory. For example, the administrative log files in **/usr/adm** grow over time but do not change in number or names.
- Dynamic directories have files added to and removed from them. The owner (at least) requires write permission to the directory in order to do this.

The second way of classifying directories is based on how open they are to general access:

- Private directories are restricted against access by all users other than the owner.

  Section 3.3, **Directory and File Security** gives information all users should know in order to protect and analyse their private directories. You should follow these safeguards in managing the system's private directories.

- Pass-through directories give search permission to all users.

  A pass-through directory gives a user access to objects in or under it, provided the user knows their names. Users cannot see the directory with *ls*, because that requires read permission as well as search permission. A pass-through directory typically holds subdirectories with different uses and permissions. It offers limited security through concealment.

- Informational directories give read permission to all users.

  Informational directories can be listed by users. Like pass-through directories, informational directories can group together diverse subdirectories. Informational directories do not conceal their contents. Typical informational directories include **/lib** and **/usr/include**.

- Public directories let anyone create or remove objects. Temporary directories such as **/tmp** and **/usr/tmp** are typical public directories. **Section 3.3.2**, **Temporary Directories** explains dangers of using public directories.

The different types of directory have the permissions listed in the following table:

---

7. Vendor documentation typically suggests actual permissions, group and owner, to apply to the directories, utility programs and data files that the **X/Open Portability Guide** defines. This varies among X/Open-compliant systems.

Permissions on Directories		
	Static	Dynamic
Private	`dr-x------(500)`	`drwx------(700)`
Pass-through	`dr-x--x--x(511)`	`drwx--x--x(711)`
Informational	`dr-xr-xr-x(555)`	`drwxr-xr-x(755)`
Public	not applicable	`drwxrwxrwx(777)`

The security of file systems can be improved by taking an inventory of all system directories (those not owned by a particular user). You should be able to classify all directories as described above. If any directory does not lend itself to classification, move the files in it to other directories until each directory is a pure example of one of the directory types. Make each directory as restrictive as is feasible.

6.5.2 File Systems

Each disk or disk pack is typically a separate file system. If you divide discs into partitions, each partition can be a separate file system. Specifying the file system and device on which each directory resides provides a number of benefits.

Enable Start-up

All files required for start-up should reside on the same file system. Files here must install the other file systems.

Protect Data From Changes

Separating frequently modified data from static data reduces the chance of an error or penetration in one part of the system affecting other parts. If you allocate data to file systems sensibly, you may find you can mount certain file systems in read-only mode, or even physically write-lock certain devices, for additional security.

Isolate Vulnerable Directories

Segregate temporary directories, such as **/tmp**, to a separate file system. Place on a separate device those directories you cannot prevent users from exhausting, such as **/usr/spool**.

Prevent Duplication of Data

Put members of the same group on the same file system. This lets them make links to each other's data or to group data without forcing the system to make copies of objects.

Improve Performance

If all the files on a file system are changed only rarely, you can operate that file system with nearly no free space and lay out the data to minimise fragmentation and increase speed.

Isolation of Corruptions

Users with different needs, such as very fast access or large quantities of storage, should be accommodated with minimal impact to other users. A file system can lose its integrity if it becomes completely full, if power fails or if the computer halts in the middle of an

operation. A corrupted file system is one where the directories, objects, block allocation tables and other attributes are not synchronised. For example, the operating system may have created an object but not yet have updated any directory to point to it. If there is a discrepancy between the objects on a disk and the block allocation tables, a block that is already in use could be used again, destroying the contents of that block.

Most X/Open-compliant systems provide a way to check the integrity of a file system, such as the *fsck* utility. The start-up script (for instance, **/etc/bcheckrc**) should invoke this program and apply it to all active file systems.

Running with a corrupted file system raises the possibility of random binary data appearing anywhere in any stored object. This seriously undermines security. Hence, once corruptions are detected, the offending file system should not be made unavailable to general users until it has been repaired. Repair may involve restoration from a backup.

Backup and Restore

Backup and restore operations on file systems are discussed in **Section 3.5.5**, **Backup and Restore**. Vendor documentation may provide helpful details.

6.5.3 Protections on Objects

The following objects must be especially protected from general use.

Kernel File

The file containing the image of the operating system is read into memory whenever the computer is restarted. This file should not be generally accessible. A penetrator who modifies the kernel file can disable access-checking or make random changes to the file that may deny service completely. Furthermore, the penetration may not become evident until the next restart of the computer.

Utilities such as *ps* read the kernel file to determine the locations of process data in varying releases of the operating system. (They typically also read **/dev/kmem** to get the actual data their caller requested.) You can achieve this while continuing to protect the file by giving such utilities the set-user-ID or set-group-ID attribute and the same owner that owns the kernel file and **/dev/kmem**.

Most systems have a file, typically called **mnttab** or **fstab**, used by the *mount* utility. It contains a list of the available file systems, and indicates which are readable and writable. If *mount* is executable by general users, then they must be able to read this file. They must not be able to write it.

/etc/utmp

Some systems implement a file **/etc/utmp** to hold login names, terminals and login times for each user currently logged in. If present, this file must be readable by all users. It must not be writable by general users.

/etc/passwd

On systems that implement user and authentication data in **/etc/passwd**, the file must be readable by all users. It must not be writable by general users.

Foreign File Systems

Section 3.5.5, **Backup and Restore** lists precautions for restoring files from backup media such as tape which can be summarised as follows. Assume the medium is insecure, check all directories for special files and privileged programs, and verify the identity of every program. The same precautions are warranted in mounting foreign file systems. First, apply a utility like *fsck* to the foreign file system to be sure it is not technically corrupted. Create a special directory, owned by the super-user and excluded to all other users by having the permissions

```
drwx------ (700)
```

Mount the foreign file system, read-only, at that location, for example, by loading the disk and typing:

```
# mount /dev/sctfdm1 /securemount -r
```

Only after making these tests can you unmount the file system and mount it again in its desired location with the proper permissions.

6.5.4 Contents of Objects

To ensure security, you must control the contents of certain objects, as well as permissions. All these objects must withhold write permission to general users. They should also withhold read permission, so that any security flaws will not be evident to a penetrator.

Privileged Programs

Privileged programs must be controlled. The only utilities that need to have the set-user-ID attribute are those that intrinsically involve security transitions and special privileges, such as *login* or *at*; and those that use device special files or otherwise unusable objects, such as *ps* or *uucp*. In the latter case, utilities may not have to be privileged; you may be able to give them sufficient access to objects by setting owner and group properly.

Privileged programs administer part of the security of the computer at large, and must have authentication and access checks at least as tight as elsewhere in the system. **Section 4.4**, **Privileged Programs** gives guidelines for writing and maintaining these programs.

Shells

Shells are inherently insecure because of their ability to invoke other programs. No shell should be a privileged program. No program that offers an escape by which users can submit shell commands should be privileged.

Shell Scripts

Shell scripts that run with privileges must be controlled. These include the following:

- **/.profile**, the login script for the super-user. (This script should also adhere to the general rules for login scripts, discussed in **Section 3.4.1**, **Profile Files**.)

- **/usr/lib/cron/.proto**, the login script for *at* jobs.
- Start-up scripts, such as **/etc/rc**, which run during recovery in order to restart time-sharing.
- *cron* scripts in the root **crontab** file. They are all assumed to run with super-user privilege. Use a command like */bin/su* in these scripts to change to the pseudo-user *uucp*, *backup* or other identity, as applicable.

All these scripts should rigorously establish their own environment. For example, they should define their own search paths, using full pathnames, and set their own *umask*.

Output from all these scripts should be captured in a log file. Start-up scripts can instead produce output on the console if the console is physically secure and is a printing terminal. You can use the *tee* utility to send a program's output to both a terminal and a log file. Review saved output from such scripts periodically, and fully investigate any failures or anomalies.

Add the following to the start of each privileged shell script to authenticate its user and report unauthorised uses via *mail:*

```
PATH=/bin:/usr/bin:/etc     # Search only specific sys directories
export PATH                 # This applies to all commands
umask 077                   # Exclude general access to all
                            # files created
reporter=auditor
LOGNAME=‘realuser‘          # (We define this program in the
                            #  next example)
#
# Report unauthorised uses of this script
#
if test $? -ne 0
  echo "User ID uncertain; program $0 not executed on ‘tty‘ \
at ‘date‘" | mail $reporter
  exit 1
elif test $LOGNAME != admin1 -a $LOGNAME != admin2; then
  echo "$LOGNAME has tried to use $0 on ‘tty‘ at ‘date‘" | \
          mail $reporter
  exit 1
#
# You may elect to report authorised uses too
#
else
  echo "$LOGNAME has successfully called $0 on ‘tty‘ at ‘date‘" |\
        mail $reporter
fi
```

The above example assumes the existence of a program called *realuser*, as in the following example. This program obtains the real (login) user name of the process. It is comparable with the *id* utility, which reports a process's effective user name.

```
/* realuser.c
 *
 * Get the process's real UID and try to find
 * the entry in /etc/passwd.
 */
#include <stdio.h>
#include <pwd.h>
#define PW_NULL ((struct passwd *) 0)
main(argc, argv)
   int argc;
   char *argv[];
{
   int ruid;
   struct passwd *pw;
   ruid = getuid();
   pw = getpwuid(ruid);
   if (pw == PW_NULL)
   {
      fprintf(stderr,
              "%s: Can't find UID %d in passwd file.\n",
              argv[0], ruid);
      exit(1);
   }
   printf("%s\n", pw->pw_name);
   exit(0);
}
```

Data Files

A file such as **/etc/inittab** may tell the *init* initial process how to create all the subprocesses, such as login shells, that will exist. This file determines, for instance, whether a restart of the computer produces time-sharing or single-user operation. Its contents must be correct to ensure security.

The files **cron.allow** and **cron.deny** must have correct contents to ensure security. This also applies to **at.allow** and **at.deny**. In each pair, the **allow** file lists the users allowed to use the respective utility, and the **deny** file lists the users prohibited from using it. It is a more secure technique to specify the contents of the **allow** file. This means that users you overlook cannot use the utility. If both files are missing, only a user with appropriate authorisation can use the utility. If only the **deny** file exists and is empty, global usage is permitted.

System Upgrades

Be sure to authenticate any upgrades (revisions of system software) that you receive. Systems have been penetrated by counterfeit revisions of system software. Be sure of the identity of persons claiming to be representatives of your hardware and software vendors.

6.5.5 Protected Subsystems

The file system consists of the root directory, ‘‘/’’ and all objects and subdirectories within it. This is the root hierarchy. A subsystem is a hierarchy that starts not at ‘‘/’’ but at some other directory that you specify. For example, each user has a subsystem: the hierarchy under the user’s home directory. Generally, users are not restricted to their own hierarchies.

The *chroot* utility and the *chroot*() function specify a directory that the process should regard as the root directory from then on. Then the path ‘‘/’’ no longer refers to the true root but to the home directory of the specified subsystem. This is a *protected subsystem*. It is a potent way of isolating the user from all resources outside the subsystem. For the duration of the process, the user cannot refer to anything outside the subsystem, even when running a privileged program or becoming the super-user, because the user has no grammatical way to refer to those outside objects. Any pathname that starts with ‘‘/’’ is translated to a path within the subsystem.

A user’s record in the user database can specify *chroot* as the initial program. This forces the user’s entire session to take place within a protected subsystem. You can instead have the login mechanism itself run within a protected subsystem, typically by specifying ‘‘*’’ as the initial program. Then the subsystem must contain sufficient objects to enable login.

You can restrict the utilities and the devices that protected subsystem members can use by not placing copies of the utilities or device special files inside the subsystem.

Always use protected subsystems when a part of the computer has to be opened for extended periods of time to persons not under control of your organisation. For example, companies offering time-sharing services often use protected subsystems. Also consider the use of protected subsystems for guest accounts.

6.6 COMMUNICATION

Modems and network interfaces expose the computer to users not necessarily under your organisation's control. Network connections are typically authenticated links to a well-defined remote computer. This reduces the possibilities of mischief. However, network data transfers may use a higher speed than telephone transfers, increasing the amount of data that may be disclosed or lost before you have time to react.

6.6.1 Communication Modes

Communication modes include cases where the computer answers the telephone, where it initiates a telephone call, or where communication media other than telephones are used.

Incoming telephone calls are handled in the following way:

- A terminal line may be attached to a modem instead of to a video or printing terminal. (**Section 6.4**, **Machine Security** describes security controls on terminal devices.) The modem plugs into a telephone outlet and permits the computer to answer incoming calls. This permits remote use of your computer from any location where there are a compatible terminal and modem. Such terminal lines have the status of shared terminals.

  In this mode of remote communication, the computer is open to any caller, authorised or unauthorised. The normal password security scheme is usually sufficient to prevent unauthorised login, but a determined penetrator can sometimes breach it.

  The following measures will reduce a penetrator's chances of breaching security by guesswork. Treat telephone numbers that your computer auto-answers as secret. If possible, change your system's tables (for example, *getty*) or the *login* utility so that the computer does not advertise its identity, or even which operating system it uses, until it authenticates the caller. Have a single telephone number to which users call. If possible, restrict use of this line to requests that the computer call back.

Outgoing telephone calls can also be made using the modem. In these cases, it is under the control of a process not logged in there, such as a daemon. The device special file (see **Section 6.4**, **Machine Security**) pertaining to the terminal line regulates who can use it.

- The *cu* (call UNIX) utility lets a user of your computer call out to a remote computer and use it as though the user's terminal were directly attached to the other computer. **Section 3.5.4**, **Remote Sessions** lists precautions that make it less likely that a penetrator of your computer could get information on breaching security of other computers.
- Most X/Open-compliant systems use a subsystem called *uucp* (for UNIX-to-UNIX copy) to transact with remote computers. The *uucp* subsystem transfers files and mail messages between computers and lets users execute commands on a remote computer. It also provides the transport services required by applications such as bulletin board systems. The *uucp* subsystem can use telephone lines and modems as the communication media, or it can use other computer-to-computer communication technologies.

6.6.2 Security Risks

Unless you take steps to secure it, *uucp* will usually let any remote user read, write or execute any object on your file system that has the respective permission (including account and password data); delete certain objects or substitute other objects; masquerade as another user or the *uucp* subsystem itself; and disrupt or disable normal network traffic.

A remote penetrator may scan files like **/etc/passwd** (for accounts without passwords) and *uucp* administration files (to find unencrypted passwords to other systems). The penetrator may try to alter accounting files to ensure future access, or may try to trick users into executing bogus programs that will give the penetrator the local user's power.

To ensure network security, you must:

- Create an account in the user database for a *uucp* administrator, and a separate account for each remote system that may log in.
- Specify which parts of the file system are available to local users and to remote systems.
- Specify which commands each remote computer can execute.
- Maintain restrictive permissions on *uucp* administrative files, spool files and device files.
- Monitor network audit and control files.
- Have hardware or firmware that detects and can force a break in a network connection.

Any time there is a security breach with unknown factors, shut down the network and telephone access to the computer and inform the network administrator. Open the computer to external access only when you have identified the cause of the breach.

The rest of this section describes ways to improve the security of the *uucp* subsystem. There are two *uucp* packages in common use: an original *uucp*, and a newer and more secure one, called HoneyDanBer.

6.6.3 uucp Accounts

Both the *uucp* administrator and remote computers are required to log in. Traditionally, the administrator uses user name **uucp**, and the remote computer uses **nuucp**. Both users are in group **uucp**.

On systems that support **/etc/passwd**, the file would typically include the following entries:

uucp:*encrypted*:5:5:UUCP Administrator:/usr/lib/uucp:/bin/sh
nuucp:*encrypted*:6:5:Remote System:/usr/spool/uucppublic:/usr/lib/uucp/uucico

We suggest the following improvements to this traditional approach:

- Give each remote computer a separate account in **/etc/passwd**. This lets you audit the frequency and duration of access of each remote computer, and give different remote computers different resources on your computer. All remote computers can be members of the **uucp** group in order to share access to the *uucp* programs.

- The *uucp* administrator (pseudo-user ‘‘uucp’’) is a sensitive security role on your computer. This user resolves network communication problems and audits network use. The network administrator needs an interactive login shell, such as */bin/sh*. As described in **Section 6.1.3**, **Pseudo-Users**, you should establish a separate account for every person serving as *uucp* administrator. These accounts are also members of group **uucp**.

  All network management programs, especially the set-user-ID ones, should be owned by the **uucp** group.

Using this method,

```
$ grep uucp /etc/passwd
```

might yield:

```
uucp1:encrypted:51:5:John Doe (UUCP Admin):/usr/lib/uucp:/bin/sh
uucp2:encrypted:52:5:Fred Smith (UUCP Admin):/usr/lib/uucp:/bin/sh
remote1:encrypted:87:5:Remote #1:/usr/spool/uucppublic:/usr/lib/uucp/uucico
remote2:encrypted:88:5:Remote #2:/usr/spool/uucppublic:/usr/lib/uucp/uucico
```

The login shell for the remote computers must be */usr/lib/uucp/uucico*. This daemon restricts the account to limited activities that you can audit. A traditional alternative to specifying the login shell *uucico* directly in **/etc/passwd** is to invoke *uucico* from **.profile.** This lets you increase the *ulimit* for some remote computers, enabling the transfer of large files. In this case, use the *trap* command of */bin/sh* to avoid interruption at any time that *uucico* does not have control of the process. Executing *uucico* from the login shell keeps it within the same process.

Here is a sample **.profile**:

```
trap "trap '' 0; exit 1" 0 1 2 3 13 15
PATH=/bin:/usr/bin
export PATH
ulimit 2000
umask 022
exec /usr/lib/uucp/uucico
exit 2
```

6.6.4 Call-Back

Both *uucp* subsystems support call-back. A field in the *uucp* definition of a remote computer specifies call-back. Whenever the local computer receives a telephone call and the remote computer successfully identifies itself as a computer to which call-back applies, the local computer hangs up the telephone and calls the preset telephone number of the remote computer. This prevents penetration since it authenticates telephone users of the local computer. However, it can only be applied in one direction; if both computers specify call-back for contact with each other, an infinite process results. One computer must accept the call from the other without the authentication that call-back provides. This called computer must provide true authentication, since any call could be from a penetrator. But if this is the computer that usually initiates transfer requests, calls from the other computer will coincide in time with its own requests for contact with the other computer. This assists in authentication.

6.6.5 Original uucp

You specify which remote computers can call your computer, and which parts of the file system are valid for file transfer by local users and remote systems, by making entries in **/usr/lib/uucp/USERFILE**. Each entry is in the format:

[login],[sys] [c] pathname ...

login is the login name of a user (local user or remote computer) defined in **/etc/passwd** on the local computer

sys is the name of a remote computer

c if present, specifies call-back

pathname is a directory or file on the local computer

The *uucico* utility compares the *login* and *sys* fields with information pertaining to a particular use of *uucp*. Either field may be null. In this case, that field matches any use of *uucp*. The *uucico* utility scans **USERFILE** from start to end. Therefore, lines with defined *login* and *sys* fields should precede lines with null fields. (Specific entries should precede generic entries.) A line with a null field applies to all sessions not covered by preceding lines. An entry with null *login* and *sys* fields applies to any session not covered by previous entries. Such an entry should only occur at the end of **USERFILE**.

Once *uucp* finds a matching entry, it restricts file transfers during that use to the one or more pathnames specified in the same entry, and stops searching **USERFILE**.

If the pathname is a directory, a transfer can involve any objects in the directory, including subdirectories (that is, access to the entire hierarchy is permitted). If the pathname is a file, a transfer can involve the named file. For example, if you specify only the name of a file, the session is restricted to that single file.

Apply appropriate permissions to any hierarchy before you list it in **USERFILE**. Do not specify user hierarchies in **USERFILE**. They are more easily changed and under less control.

Inbound Access Control

The *uucico* login shell starts by obtaining the name of the remote computer and the login name under which *uucico* is running (often **nuucp**). It searches **USERFILE** for an entry describing this combination and obeys the restrictions contained in that entry.

If each remote computer has a separate account, the *login* and *sys* fields in **USERFILE** are both specific to one remote computer.

USERFILE should begin with such specific entries, explicitly specifying both the *login* and *sys* fields. You may elect to follow these fields with one generic entry, omitting both *login* and *sys* fields. This entry applies to any remote sessions not covered by specific entries. It also applies to any transfers initiated by a local user.

If no entry in **USERFILE** applies to the current use of *uucp*, and **USERFILE** does not contain a generic entry, then the transfer cannot occur.

Outbound Access Control

If an entry in **/usr/lib/uucp/USERFILE** specifies a local user, then the pathnames in that entry restrict the files the user can send to a remote computer. (A line with a null username restricts all users not named in **USERFILE**.)

Such entries hinder users without providing any additional security. The user is still free to use *cp* to copy any file with read permission into the directory specified in **USERFILE**, and transmit the file from that directory.

6.6.6 HoneyDanBer uucp

The HoneyDanBer *uucp* subsystem follows the same basic protocol as the original package, but gives you more detailed control over network operations. This makes it more secure. The HoneyDanBer *uucp* also separates the spool control files into directories according to function (errors, connection attempts, session logs, lock files and archives) and, in some cases, into separate files for each machine.

The **/usr/lib/uucp/Permissions** file has *LOGNAME* and *VALIDATE* options that define the names of remote computers and require them to use particular accounts to log into your computer. A typical line in **Permissions** includes at least these fields:

```
LOGNAME=local_account VALIDATE=remote_system_name
```

The **Permissions** file is crucial to network security. It must not contain errors. Use the *uucheck* utility after every edit to **Permissions** to view the resulting settings in a readable form. Type this command:

```
$ /usr/lib/uucp/uucheck -v
```

This utility also checks whether the directories themselves exist and whether they contain other files.

An important security feature of the HoneyDanBer *uucp* is that it can take user-defined action when an apparent penetration occurs. If a remote computer not listed in **/usr/lib/uucp/Systems** calls your computer, and if a **/usr/lib/uucp/remote.unknown** file exists and is executable (use the permissions

```
r-x------ (500)
```

and owner **uucp**), then *uucp* will not only deny access, but also record the event in the appropriate log file in **/usr/spool/uucp/.Log** and execute **remote.unknown** as a shell script.

An entry in **Permissions** can include additional options:

CALLBACK specifies call-back (see **Section 6.6.4**, **Call-Back**)

COMMANDS lists the commands the remote user is allowed to execute (see **Section 6.6.8**, **Remotely Executable Commands**)

[NO]READ determines whether or not the remote computer can extract data from the local computer

REQUEST lets the local computer respond to requests initiated by the remote computer (for transfers in either direction)

SENDFILES lets the local computer respond to requests initiated by a local user (for transfers in either direction)

[NO]WRITE determines whether or not the remote computer can store data at the local computer

These options are more readable and give more precise control than the **USERFILE** file in the original *uucp*. The *REQUEST* and *SENDFILES* options let you manage remote computers that tend to exchange large volumes of data when the local computer calls it (and you are paying for the telephone call).

Specify settings explicitly instead of using default settings. Specify different machines on different lines of the **Permissions** file. Then changing one system's permissions will not implicitly change those of other systems. Promptly remove obsolete entries from the file.

6.6.7 Remote Logins

Information on how to phone out to remote computers appears in certain files. Unauthorised disclosure or changes to these files threaten the security of the entire network. There is a **lines** file (called **L.sys** in the original *uucp*, or **Systems** in HoneyDanBer), and a **dialcodes** file (called **L-dialcodes** in the original *uucp*, or **Dialcodes** in HoneyDanBer). All files reside in the **/usr/lib/uucp** directory. Both *uucp* subsystems use the same file formats.

Each entry in the **lines** file contains the telephone number, login and password information for one remote computer.

Some computers may appear on several lines if there are several ways to reach it. The local computer tries each method, in the order listed, until a method succeeds.

Although a penetrator who succeeded in contacting a remote computer would get *uucico* as a login shell, the penetrator might be able to penetrate the remote computer further by guessing or stealing passwords for that computer. By using a computer that simulates the *uucp* protocol, the penetrator can also masquerade as your computer.

You can increase network security by separating the telephone number from the rest of the login information. Instead of placing telephone numbers in the **lines** file, use symbolic names. Place lines in the **dialcodes** file to associate each symbol to an actual telephone number. For example, in place of this entry in the **lines** file:

```
remote1 Any ACU 300 5551234 ogin:-@-ogin: uumyname word: xyzzy
```

you can use a symbolic name for the telephone number:

```
remote1 Any ACU 300 phone ogin:-@-ogin: uumyname word: xyzzy
```

Define the name in the **dialcodes** file:

```
phone  5551234
```

6.6.8 Remotely Executable Commands

In the original *uucp*, you typically specify any commands you want to withhold from remote users by editing the source of the *uuxqt* utility. Some advanced varieties of the original *uucp* take this data from a file. But even these implementations do not let you differentiate among remote computers.

In the HoneyDanBer *uucp*, you use the *COMMANDS* option in **/usr/lib/uucp/Permissions** to allow certain commands to be executed. You can specify full pathnames, single command names (subject to the applicable search path) or the value ALL. Explicitly specify *COMMANDS* for each remote computer. Always use the *VALIDATE* option in conjunction with *COMMANDS* to be sure no commands are available until the remote computer properly identifies itself. You can use the keyword ALL in conjunction with some explicit commands. The effect of this is that the specified commands are permitted, and the appropriate search paths are used for other commands.

6.6.9 Permissions

Vendor documentation specifies permissions, owner and group for *uucp* files. In the original *uucp*, the utilities *uucp*, *uulog*, *uuname*, *uustat*, *uux*, *uucico*, *uuclean* and *uuxqt* are owned by *uucp* and have the set-user-ID attribute. You should tightly restrict access to them. The HoneyDanBer *uucp* has a greater number of files. Each file can carry more precise permissions. Administrative files reside in the **/usr/lib/uucp** hierarchy, spool files are in **/usr/spool/uucp** and programs are in **/usr/bin**.

6.6.10 Breach Detection

To detect attempts to breach security, scan all log files using *uulog*. Look for attempts to gain access to sensitive files. Failed attempts may mean a remote user is probing your system.

In the original *uucp*, the **LOGFILE** file records both successful and unsuccessful attempts at system connections, file transfers and remote command execution. The **/usr/spool/uucp/SYSLOG** file records successful file transfers. Scan it to determine the types and sizes of files transferred and the users initiating the transfers. Include **LOGFILE** and **SYSLOG** in the list of security-related log files used in auditing. Protect the **/usr/spool/uucp/SEQF** file as tightly as you do **LOGFILE** and **SYSLOG** because it provides the unique sequence stamp used by them. They should be used by *uulog* and other *uucp* utilities only, and not by general users. In the HoneyDanBer *uucp* subsystem, the filenames are different and there are additional administrative programs that produce reports. There is a log file in the **/usr/spool/uucp/.Log** directory for every authorised remote computer. The *uudemon.cleanup* script removes old data from this directory, placing it in the **/usr/spool/uucp/.Old** directory. The *uustat* utility provides readable information on work that is queued (with the *-q* option) and communications in progress (with the *-p* option).

Appendix A

Security-Related Utilities and Files

A.1 SECURITY-RELATED UTILITIES

Utility	Description
acctcom	display audit information
chgrp	change file group owner
chmod	change file access permissions
chown	change file owner
chroot	change root directory to restrict user environment
crash	system debugger (restrict to super-user only)
crontab	handle user crontab file
crypt	encrypts/decrypts data
ed -x	editor: encrypts/decrypts data
find	walk file system tree (may be used by security audit)
fsdb	file system debugger (restrict to super-user only)
id	display effective and real UID and GID
login	allow user to login
logname	get login name of owner of current process
makekey	output 2-byte salt and 11-byte key using DES algorithm
mknod	create an ordinary, directory or special file
ncheck -i	list file name(s) associated with i-node
ncheck -s	list set-user-ID and special files
newgrp	change GID
passwd	change login password
ps	display list of active processes
red	restricted editor: encrypts/decrypts data
rsh	restricted shell
su	change effective and real UID to that of another user
umask	set file-creation mask
vi -x	editor: encrypts/decrypts data

A.2 SECURITY-RELATED FILES

Special File	Description
/dev/console	system console (should not be writable to others)
/dev/kmem	kernel memory (should not be accessible)
/dev/mem	system memory (should not be accessible)
/dev/sctfd*	floppy disk special files
/dev/sctmt*	tape special files
/dev/tty?*	terminals (should not be writable to others)

File	Description
/etc/group	group file (passwords encrypted)
/etc/passwd	password file (passwords encrypted)
/etc/utmp	current status of login activity
/etc/wtmp	log of login activity

Library	Description
/lib/libc	C-language library with *crypt*
/lib/libp/libc	profiling C-language library with *crypt*

File	Description
/usr/adm/*acct*	command audit trail
/usr/adm/sulog	log of *su* activity
/usr/adm/wtmp	login session audit trail
/usr/lib/uucp/L.cmds	controls UUCP remote execution
/usr/lib/uucp/L.sys	contains unencrypted UUCP login information
/usr/lib/uucp/Permissions	controls UUCP access to filesystems
/usr/lib/uucp/USERFILE	controls UUCP access to file system
/usr/spool/uucp/LOGFILE	log of UUCP activity
/usr/spool/uucp/SYSLOG	log of UUCP file traffic

A.3 SECURITY-RELATED FUNCTIONS

Function	Description
chmod ( )	change file access permissions
chown ( )	change file owner and / or group owner
chroot ( )	change root directory to restrict user environment
creat ( )	create file with specified access permissions
crypt ( )	generate an encrypted password
cuserid ( )	get login name
encrypt ( )	encrypt / decrypt data
fstat ( )	obtain status information for an open file
ftw ( )	walk file system tree (may be used by security audit)
getgrgid ( )	get group entry from group database for specified GID
getgrnam ( )	get group entry from group database for specified group name
getegid ( )	get effective GID of process
geteuid ( )	get effective UID of process
getgid ( )	get real GID of process
getlogin ( )	get login name of current process
getpass ( )	display prompt and read password (with echo suppressed)
getpwnam ( )	get entry from user database for specified login name
getpwuid ( )	get entry from user database for specified UID
getuid ( )	get real UID of process
link ( )	make a new path to an existing file
mkdir ( )	make a directory
mkfifo ( )	make a FIFO special file
setgid ( )	set effective GID (root sets real GID also)
setkey ( )	set the key for subsequent use with *encrypt* ( )
setuid ( )	set effective UID (root sets real UID also)
sigaction ( )	examine and change signal action
signal ( )	specify action to be taken on receipt of a specified signal
stat ( )	obtain status information for a file
umask ( )	set file-creation mask
unlink ( )	remove (a path to) a file

Index