NIST Special Publication 800-7

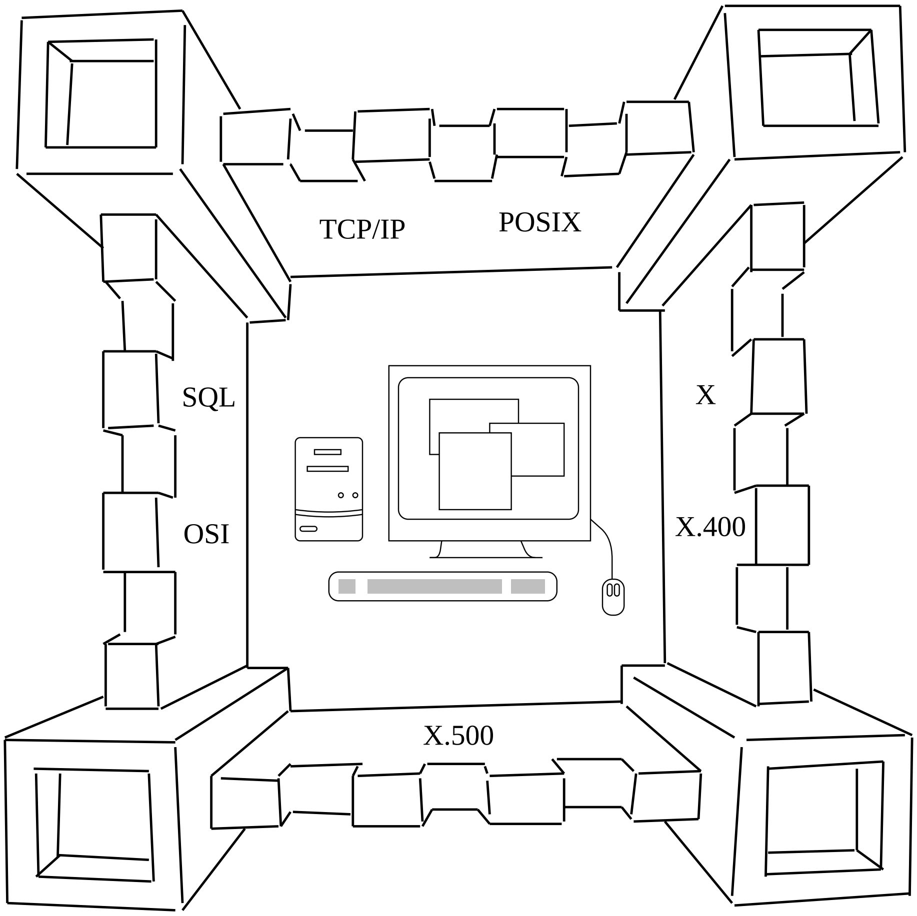
Computer
Systems
Technology
U.S. DEPARTMENT OF
COMMERCE
National Institute of
Standards and
Technology

# Security in Open Systems

Robert Bagwill
John Barkley
Lisa Carnahan
Shu-jen Chang
Richard Kuhn
Paul Markovitz
Anastase Nakassis
Karen Olsen
Michael Ransom
John Wack

John Barkley, Editor

COMPUTER SECURITY



# Contents

# List of Figures

# List of Tables

Because of the nature of this report, it is necessary to mention vendors and commercial products. The presence or absence of a particular trade name product does not imply criticism or endorsement by the National Institute of Standards and Technology, nor does it imply that the products identified are necessarily the best available.

# Preface

The Public Switched Network (PSN) provides National Security and Emergency Preparedness (NS/EP) telecommunications. Service vendors, equipment manufacturers, and the federal government are concerned that vulnerabilities in the PSN could be exploited and result in disruptions or degradation of service. To address these threats, NIST is assisting the Office of the Manager, National Communications System (OMNCS), in the areas of computer and network security research and development. NIST is investigating the vulnerabilities and related security issues that result from the use of open systems platforms, i.e., products based on open standards such as POSIX and OSI, in the telecommunications industry.

This report is intended to provide information for the practicing programmer involved in development of telecommunications application software. In short, it provides answers to the question "How do I build security into software based on open system platforms?" It is not intended to be tutorial in nature and assumes some knowledge of open systems and Unix. Many of the references cited are tutorial and may be used to obtain any background information required.

For each topic in open system security, the goal of this report is to locate in one place the most informed exposition possible for that topic. Consequently, this report is the result of the efforts of several individuals who possess the expertise required to author the various chapters. The author(s) of each chapter is identified after the chapter title.

# Part I

# Open Systems

# Chapter 1

# Introduction

**Richard Kuhn**

The public switched network (PSN) provides services that are essential to U.S. citizens and government agencies alike. Disruption of telecommunications services would clearly represent a serious threat to public safety and security. A 1989 report of the National Research Council (NRC), "The Growing Vulnerability of the Public Switched Network" [Cou89], outlined the concerns of the government for maintaining the integrity of the PSN against intruders. A report the following year by the President's National Security Telecommunications Advisory Committee (NSTAC) concluded that "until there is confidence that strong, comprehensive security programs are in place, the industry should assume that a motivated and resourceful adversary, in one concerted manipulation of the network software, could degrade at least portions of the PSN and monitor or disrupt the telecommunications serving [government] users" [Cou90]. In addition, outages experienced by telecommunications providers in the recent past have focused the federal government's attention on the need to ensure dependable communications.

In the past, there were relatively few telecommunications providers, and their products were built on proprietary platforms. The Federal Communication Commission's Open Network Architecture (ONA) requirements specify unbundled and equal access to the PSN for Bell Operating Companies and their enhanced services competitors [Com86]. The National Research Council notes that ONA can increase network vulnerability in two ways:

> First, ONA increases greatly the number of users who have access to network software. In any given universe of users, some will be hostile. By giving more users access to network software, ONA will open the network to additional hostile users. Second, as more levels of network software are made visible to users for purposes of affording parity of network access, users will learn more about the inner workings of the network software, and those with hostile intent will learn more about how to misuse the network [Cou89, p. 36].

Greater network access is changing the telecommunications industry to one where many third party service providers are building products that must work with products from other companies [Dol88], [Sim88], [SH88]. This new telecommunications environment has been characterized as one with: a large number of features; multi-media, multi-party services; partial knowledge of the feature set by service designers; lower skill and knowledge levels of some service creators; multiple execution environments from different vendors; and distributed intelligence [Dwo91]. As noted in the NRC report, some fraction of those with access to the network must be assumed to be hostile. Those with hostile intent may include employees of telecommunications service providers.

Like most of the computer industry, both the Bell Operating Companies and third party service providers are moving toward use of standards-based, open system products to reduce costs and improve interoperability and portability of their products. For example, one Bell Operating Company is revising its operations center computing support to "transition the existing networks to use the ISO Open System Interconnection (OSI) based network and the common network services that are independent of specific computing and application environments [bel90]." Computing systems based on standard interfaces such as OSI are referred to as "open systems."

Beginning with the ISO OSI and the IEEE POSIX operating system interface standard, a great many open systems standards are beginning to appear, and open systems products are being provided by every major computer vendor. In short, an open system standard is an interface specification to which any vendor can build products [Kuh91]. There are two important points. First, the specification simply defines an interface. For example, although POSIX is derived from UNIX, non-UNIX operating systems such as Digital's VMS can also provide a POSIX interface. Second, the specification is available to any vendor and evolves through a consensus process that is open to the entire industry.

Until now, users were often "locked in" to products from a particular vendor because their applications would run only on that vendor's operating system. The move to open systems will reduce this dependence. Application systems will increasingly be built on products from a variety of vendors. But many needed standards are not complete, and some non-standard functions will always be needed because standards must necessarily lag innovations in technology. Organizations must build applications from both standard and non-standard components. In addition, the inherent limitations of software testing make it likely that many "standard" components will have subtle incompatibilities.

The term "open" thus applies to two different aspects of the telecommunications environment: the FCC's ONA requirements that allow multiple vendors to have equal access to the network; and the open system platforms based on standards, such as POSIX and OSI, that are used in building computer based applications for the new open telecommunications environment. It may be easier for intruders to attack a system whose behavior is standardized and well known, or which shares common flaws with other systems built on the same standards. A Bellcore report found that "intruders were assisted in their endeavors by the openness and standardization that the telecommunications industry has undergone in the last decade [Klu92]." Security is thus a vital concern with open systems.

This report was prepared to help service designers use standard, open systems platforms in building security into their software applications. Security in an open system environment may be affected by the need to use both standard and non-standard components, and by the possibility for incompatibilities among products that claim to meet the same standard. The large number of third party service providers whose products must work together may severely complicate efforts to ensure dependability and security of the PSN. Software developers who are challenged with building applications on open system platforms will be faced with questions such as the following:

- What services and functions are available for providing authentication, access control, and similar security functions?
- What conflicts arise when using products that conform to open system standards?
- What are the limitations of these standards, and where is it necessary to supplement standard services with custom software?
- How do I build security into an open system application?

This report is intended to aid software developers who are building telecommunications applications on open systems platforms. It is designed to help programmers understand the open system environment and to use open system services in building secure applications. The next chapter introduces the IEEE POSIX Open System Environment, which is centered on POSIX and OSI standards. Following this introduction to open systems, four parts of the document explain security features for four main categories of open system services: operating system services, human/computer interface services, data management services, and network services. It is not possible to describe all of them in this document. Furthermore, not all have security features (e.g., most programming languages). The approach taken in this document is to describe the most important standards in each category. In addition to *de jure* standards from IEEE and ISO, some *de facto* standards such as the Kerberos authentication protocol and the X Window System are also discussed. These are included because they are almost universally available on POSIX based systems, and because they provide critical functions that may not have counterparts in formal standards.

Although this document is intended to provide technical information for programmers, introductory material is included that should be of value to product planners, administrators, users, and management personnel who are interested in understanding the capabilities and limitations of open systems.

# Chapter 2

# The POSIX Open System Environment

**Richard Kuhn**

One of the goals of the Open System Environment (OSE) is a set of standards and public specifications designed to provide software portability and interoperability. IEEE POSIX standards serve as the basis of the OSE, with related standards such as the Open Systems Interconnection (OSI) communication supplementing POSIX to provide a complete, standards-based computing environment.

## 2.1 Open System Standards

In short, an *open system standard* is an *interface specification* – a specification that describes services provided by a software product – to which any vendor can build products. There are two important points. First, the specification is available to any vendor and evolves through a consensus process that is open to the entire industry. Second, the specification defines only an interface, so different vendors can provide the standard interface on their proprietary operating systems (see fig. 2.1).

Open system standards will make it possible to develop standard software components that can be implemented on a wide variety of hardware, making a software components industry economically practical. But, open system standards do not solve all problems associated with building interchangable software components. Software designers need to understand the capabilities and limitations of software standards, and how to deal with these limitations. This article describes important open system standards and explains how they can be used to build portable, interoperable application software components.

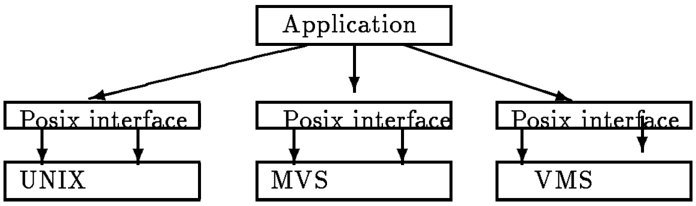



Figure 2.1: Open System Approach.

## 2.2 Interoperability and Portability

There are two important aspects to open systems: *interoperability* and *portability* [Fis93]. Interoperability refers to the capability for applications running on different computers to exchange information and operate cooperatively using this information. Portability refers to the capability for software to run on different types of hardware. Portability can further be broken down into binary portability and source code portability. Binary portability makes it possible to move an executable copy of a program from one machine to another. Source code portability requires a program to be recompiled when moving from one machine to another. The development of portable application software components depends on portability standards. Interoperability standards are necessary but not sufficient for a complete open systems environment. Software systems that are built on standards for portability and interoperability are called *open systems*.

A good example of interoperability is provided by the X Window System [RS89] protocol, which specifies how graphics primitives can be communicated between an application program and graphics software running on a workstation. An X Window application running on an IBM workstation can interact, for instance, with a user sitting at a Sun workstation. The ISO Open Systems Interconnection (OSI) standards [iee86] [Ros90] also promote interoperability. The OSI reference model defines a structure, or *reference model* for data communication standards. The reference model defines seven layers of communications system components, from the physical layer at the bottom to the application layer at the top. The model describes how components communicate with each other; i.e., it is a model for interoperability. Open system standards for application portability do not "fit in" to the OSI reference model; they are, however, complementary to data communication standards. Communication standards define communication services, but open system applications require a standard way to use those services.

Binary portability specifications are designed to provide software portability at the object code level. For example, the IBM PC hardware interface can be regarded as a de facto standard for binary portability. Executable copies of software can run on PC clones from many different manufacturers. Another example is the Application Binary Interface for systems based on the Sun SPARC processor. This specification for workstations makes it possible to move executable programs between different makes of workstation as easily as programs can be moved between different IBM PC clones. Binary portability is more difficult to achieve than source code portability, because it places constraints on hardware. Standards efforts have concentrated on developing interfaces for source code.

Open system standards for source code portability define interfaces available to application programs for services such as timing functions, security features, database access and many other essential functions. Standards could be defined by cooperatively developed source code, but within IEEE and other organizations the preferred approach has been to specify interfaces and let vendors develop competing implementations. Thus, two different operating systems may provide the same services, but one may have better performance or fault tolerance characteristics than the other. The application program interface may be specified in terms of a set of procedure calls for a particular programming language, or a language-independent specification may be accompanied by procedure calls for one or more programming languages.

The best-known standards for operating system services are the POSIX standards being defined by the IEEE Technical Committee on Operating Systems (TCOS). (The acronym POSIX is derived from Portable Operating System Interface, with an "X" to denote its UNIX origin.) Beginning with the POSIX System Application Program Interface (or kernel) standard (IEEE 1003.1-1988), IEEE has been developing a comprehensive set of standards for application portability. In May 1992, the POSIX effort comprised 20 working groups developing 34 projects.

The POSIX efforts (1003.x) have been supplemented with projects to develop standards for application interfaces to services such as windows (1201.1) and X.400 message handling (1224), that are useful on non-POSIX systems. Other open system standards have been developed through the American National Standards Institute (ANSI), the International Organization for Standardization (ISO), and other organizations. Many of these other specifications have been combined with the developing IEEE standards to define an open systems environment using the POSIX interface standards as the basis.

## 2.3 The POSIX Open Systems Environment

No single standard provides all the functionality needed by a modern computing environment. Portability and interoperability require a comprehensive set of standards. The POSIX Open Systems Environment (OSE) being put together by IEEE TCOS working group P1003.0 [POS92a] provides a standard set of interfaces to information system building blocks, covering both portability and interoperability standards. Not all of the specifications in the POSIX OSE are IEEE POSIX (1003.x) standards. POSIX functions serve as a basis, supplemented by other applicable open systems standards.

Two types of standard interfaces are specified in the POSIX OSE: the Application Program Interface (API) and External Environment Interface (EEI). The POSIX OSE Reference Model, shown in figure 2.2, shows the relationship of these interfaces to the other parts of the computing environment.

$\boxed{\text{Application Software}}$

$\Uparrow$

Application Program Interface (API)

$\Downarrow$

$\boxed{\text{Application Platform}}$

$\Uparrow$

External Environment Interface (EEI)

$\Downarrow$

$\boxed{\text{External Environment}}$

Figure 2.2: POSIX OSE Reference Model.

The External Environment refers to external entities with which the application platform exchanges information, including both the human end user, hardcopy documents, and physical devices such as video displays, disks, printers, and networks. External environment interfaces mainly provide for interoperability. EEI standards take the form of communication protocols, record and document formats, and display formats. The application program interfaces in the POSIX OSE are source code interfaces, generally in the form of programming language procedure calls, to the application platform, which is the operating system and hardware. By specifying a standard set of procedure calls, an API provides source code portability.

## 2.4 The NIST Application Portability Profile

A profile is a collection of specifications developed to meet a set of requirements. Elements of a profile may consist of either formal standards, i.e., those developed within a voluntary standards organization such as ANSI or IEEE, or de facto standards, i.e., those accepted within the marketplace. Each element of a profile may be a specification in its entirety or a specification with certain options or parameters chosen.

The NIST Application Portability Profile (APP) [Fis93] was developed by NIST in order to meet the requirements of Federal Agencies for an OSE. A Federal Agency uses the NIST APP to develop profiles specific to its individual requirements. The NIST APP is organized into several service areas which reflect the breath of services needed by an OSE application. These service areas are:

1. Operating System Services: those services providing basic manipulation of a system's fundamental resources such as processes and files.

2. Human Computer Interface Services: those services providing for the interactions between the end user and the system such as window management and multimedia access.

3. Software Engineering Services: those services supporting the application programmer such as programming languages and software development tools.

4. Data Management Services: those services providing for the definition and manipulation of data such as schema definition and query languages.

5. Data Interchange Services: those services which provide common representations for the exchange of data between systems such as document formats and display representations.

6. Graphics Services: those services providing for the creation and manipulation of displayed multidimensional images.

7. Network Services: those services providing interoperability among systems such as communication protocols.

This document is organized according to APP service areas. Each part of this document corresponds to one of the service areas defined in the APP. Not all specifications in the APP are discussed in this document and some technologies not included in the APP are discussed in this document. The decision to include a discussion of the security aspects of a technology or specification in the document is based on the relevance of the technology within an OSE and on the presence of a significant security concern associated with the technology.

# Chapter 3

# Functional Requirements Specifications for Computer Security

**John Barkley**

There are several publications available which specify computer security functional requirements in the form of evaluation criteria for secure systems. Among these are the Trusted Computer System Evaluation Criteria (TCSEC or "orange" book), the Canadian Trusted Computer Product Evaluation Criteria (CTCPEC)[CTC93], and the Information Technology Security Evaluation Criteria (ITSEC). As implied by their names, the goal of these documents is to specify a standard set of criteria for evaluating the security capabilities of systems.

As described in sections 2.1 and 2.2, a goal of open system standards is to promote the portability and interoperability of applications. This Chapter explores the ways that functional requirement specifications for computer security and open system standards complement each other.

## 3.1 Example Specifications

Within the international computer security community, much effort has been devoted to the creation of standards which permit the security functionality of systems to be evaluated. In itself, this is a large and complex problem. Within government and industry, computer security needs form a wide spectrum.

The first publicly available specification for computer security requirements was the Trusted Computer System Evaluation Criteria (TCSEC) [TCS85], first published in 1983. Although developed by the National Security Agency (NSA) to meet the needs of the United States Department of Defense, the TCSEC has been and continues to be influential in the development of commercial products and later computer security functional specifications. There are several other computer security publications directly related to the TCSEC. Among these are the Trusted Network Interpretation (TNI or "red" book) [TNI90] and the Trusted Data Base Management System Interpretation (TDI or "lavender" book) [TDI91]. The TNI shows how the criteria from the TCSEC applies to a network environment. The TDI shows how the TCSEC applies to data base management systems. The TCSEC and its related publications are often referred to as the "rainbow" series.

The Information Technology Security Evaluation Criteria (ITSEC) [ITS91] was created in a joint effort by Germany, France, the United Kingdom, and the Netherlands. Originally published in 1990, the ITSEC was developed to more completely meet the needs of those organizations handling unclassified information.

In order to further meet organizational needs for handling both classified and unclassified information, the Federal Criteria for Information Technology Security (Federal Criteria or FC) [FC92] was developed as a joint project by the National Institute of Standards and Technology (NIST) and NSA. The Federal Criteria is to be replaced by the Common Criteria now under development by NIST, NSA, Canadian and European representatives. The Federal Criteria, the ITSEC, and the CTCPEC are being used in developing the Common Criteria. The Common Criteria is to be published as a Federal Information Processing Standard (FIPS).

## 3.2 Relationship to Open Systems

Each of the specifications described in section 3.1 is a requirements specification. That a product (referred to as an "Information Technology (IT) Product" in computer security requirement specifications) satisfies some level of computer security criteria does not imply that such a product supports open system specifications. Such a product may be completely proprietary and fully satisfy computer security requirements. The goal of the developers of the specifications in section 3.1 is to insure computer system security.

Thus, for example, a program written using an API to access security features on one product that meets the Commercial Security 2 (CS2) protection profile of the Federal Criteria may or may not be portable to another CS2 product. The user interface to a utility required on a CS2 product may or may not be the same user interface to the same utility on another CS2 product. The display device used for access on a CS2 product over a communication channel may or may not interoperate with another CS2 product because the communications channel protocol used by one CS2 product differs from another.

The goal of open system standards is to insure portability and interoperability. Every open system specification was developed in response to a need by some user community. Once such a need is identified, functional requirements are developed in order to more precisely describe the needs a particular open system standard is to satisfy.

As a result of the specifications in section 3.1, the computer security community is reaching a level of consensus on requirements for computer security. The question arises as to how well products based on open system standards can meet these requirements. It is important to note that computer security functional requirements, such as those in section 3.1, are intended to be applied to implementations, i.e., products not open system specifications.

With regard to a product based on open system specifications, computer security criteria would be applied to that product's implementation, not to the collection of open system specifications which that product supports. Nonetheless, the question can still be asked of the open system specifications themselves with the caveat that it is their implementation on which final evaluation is made.

This question of how well open systems can meet computer security criteria is not a question that has just recently been asked. While the primary goal is computer security, many of the developers of computer security criteria participate in the development of open system standards intended to support existing computer security requirements. POSIX.6 (see ch. 4) is one example of such an open system standards development activity.

NIST is interested in exploring the problems of quantifying how well open system standards meet consensus computer security functional requirements [PI93]. Before such a quantification can be made, several questions must be addressed.

One question is which computer security requirements specification should be used. While generally the same in intent, each has its own form and structure. It is possible that a system which satisfies some level of one specification may not satisfy any single level of another but may satisfy requirements from several levels. Because the Common Criteria will have the most flexible framework, users can choose a level to exactly meet their requirements. Consequently, the Common Criteria, the FIPS that will be developed by NIST and NSA, is the specification of choice for Federal Agencies and is intended to be useful to organizations in the private sector.

A difficult question is what does it mean for an open system standard to meet computer security functional requirements. For most functional requirements, the answer to this question is not immediately obvious. For example, consider a requirement for a utility to examine audit trails. An API to audit trail information which standardizes access by a program could be developed. As a result, portable utilities which satisfy the requirement could be developed. In addition, a standard user interface to the utility could be developed so that users would know how to interact with the utility on any secure product which satisfies the requirement. If audit information is generated in a network environment and transmitted to a central location, a standard protocol for the communication of audit information could be developed.

Once it has been determined what it means for an open system standard to meet given computer security functional requirements, it must be determined that there is a need for such an open system standard assuming one does not already exist. Consider again the example of a requirement for a utility to examine audit trails. It may be the case that a standard user interface to an audit trail utility in not necessary because such utilities are always developed with window-based desktop displays in which user access is self-evident. Consequently, the question of a user being able to interact with any such utility is moot. In the final analysis, the community of interest will likely determine the need for an open system standard to meet computer security requirements.

Another question is, of existing open system standards, which ones are applicable to be compared against computer security criteria. In order to answer this question, it must be determined what products are addressed by a particular computer security functional requirements specification.

Thus, while the Common Criteria can obviously be applied to products which support POSIX.1 and POSIX.6, it is not inappropriate to consider the application of the Common Criteria to a product supporting an open system specification such as SQL. SQL is certainly concerned with the manipulation of resources by named users and the protection of such resources against unauthorized access. On the other hand, with regard to the Common Criteria, it is not clear how computer security functional requirements would apply to a product which supports a protocol specification, such as, OSI Transport. Each existing open system standard should be considered a candidate to be compared against computer security functional requirements. One possible way to deal with this question is to look at sets or profiles of open system specifications and apply computer security criteria to these instead of individual specifications.

Clearly, there is much work to be done to answer these questions. Until these questions are answered, it is not not clear how well the security aspects of the open system specifications described in this report meet the functional requirements of computer security evaluation criteria such as the Common Criteria.

# Part II

# Operating System Services Security

# Chapter 4

# POSIX Security Interfaces and Mechanisms

**Lisa Carnahan**

## 4.1 Introduction to POSIX Security

POSIX is a family of standards designed to ensure portability of application programs across hardware and operating systems. These standards are the products of the IEEE Technical Committee on Operating Systems, P1003 Committee. For the purpose of this discussion, two of the standards produced by this committee are of primary interest:

- **Posix System Application Program Interface Standard (ISO/IEC 9945-1: 1990)** [ISO90a]- This is the base POSIX standard. It defines the basic system services and mechanisms (e.g., input/output services, process environment, etc.) and the system calls that provide the interface between application programs and those system services. This standard has been adopted by the National Institute of Standards and Technology (NIST) as Federal Information Processing Standard (FIPS) 151-2 [FIP93b]. It will be referred to as the "POSIX.1 standard."

- **Posix System Application Program Interface - Amendment: Protection, Audit and Control Interfaces (IEEE P1003.6.1)**[1][POS92b] - This standard specifies mechanisms and interfaces to security functionality not provided in the base (i.e., POSIX.1) standard. The general areas that are covered are:

  1. discretionary access control.
  2. audit trail mechanisms.
  3. privilege mechanisms.

[1]This specification is now known as P1003.1e.

4. mandatory access control.
5. information label mechanisms.

The Protection, Audit and Control Interface Standard will be referred to as the "POSIX.6 standard."

Like the POSIX.1 standard, the POSIX.6 standard was originally grown out of work begun in /usr/group, now known as Uniforum. As the POSIX.1 standard was moved out of /usr/group and into IEEE, some security professionals within /usr/group saw the need to:

1. provide portable applications those interfaces necessary to utilize security relevant information.
2. improve on the security mechanisms that were being defined in the POSIX.1 standard.

Realizing that many users would feel that the security mechanisms defined in the POSIX.1 standard would be sufficient for their needs, it was decided that a set of security mechanisms and interfaces would be developed and placed as extensions to the POSIX.1 standard. From a security viewpoint, the improvement over the POSIX.1 security mechanisms is substantial.

The POSIX.6 Security Mechanisms address five areas of functionality:

- audit trail mechanisms,
- discretionary access control,
- information labels,
- mandatory access control,
- privilege.

According to the POSIX.6 draft, each option defines new functions, as well as security-related constraints for the functions and utilities defined by the other POSIX standards. The addition of these mechanisms to the POSIX.1 standard allows "general purpose" applications to take advantage of the security enhancements while maintaining portability. In addition, these areas are widely used by "trusted" programs - thus allowing for application portability of trusted programs.

These areas were chosen because it was felt that they encompass the de facto required areas for security in today's POSIX environments. While all these areas may not be required on a single system, some combination of them should be. Access control should be required on any multi-user system. The POSIX.6 standard supports a mechanism that allows the generation of an audit trail that can later be analyzed by an audit analysis tool. The use of audit on POSIX systems is highly encouraged. More and more users are discovering the many benefits of using information labels on their systems. The POSIX.6 standard supports these as well.

The POSIX.6 interfaces are positioned between the application system calls and the operating system. In this way the application is buffered from having to know the internals, formats, etc. that make systems unique. An application can request to know the mandatory access control label of a file without having to know where or how the label is stored internally. This is what makes application portability, and POSIX.6 focuses on providing application portability on systems that make use of the POSIX.6 mechanisms.

The standard that provides the POSIX.6 interfaces and mechanisms is currently in the balloting process. The DRAFT Standards P1003.6.1/D13 and P1003.6.2 [2] [POS92c] were the current documents at the time of this writing. It should be realized that with any standard that is cycling through a balloting process, changes to the standard may occur. Therefore, differences (hopefully slight) may arise between the information presented here regarding specifics of the standard, and the final specifications of the standard upon final approval.

## 4.2 Posix Security Functionality

### 4.2.1 FIPS 151-2 Security Mechanisms and Interfaces (P1003.1)

The POSIX.1 standard (FIPS 151-2/P1003.1) does provide some security functionality. The security functionality supported includes Discretionary Access Control using a permission bit mechanism and Privilege using a privilege mechanism. It was the intent of the POSIX.6 standard to extend this functionality and add areas of functionality not addressed by the POSIX.1 standard. An application that is POSIX.1 compliant should run successfully on a POSIX.6 compliant system.

The POSIX.1 standard supplies only a subset of the functionality supplied by the POSIX.6 standard. The functionality provided by the POSIX.1 standard should be the minimal acceptable requirement of security functionality for any multi-user system requiring POSIX-like interfaces. Whether the extended and additional POSIX.6 functionality is needed should be determined based on the security requirements of the system.

### 4.2.2 Data Structures and the Interface Scheme

The different data structures that are defined by the POSIX.6 mechanisms (access control lists, privilege attributes, mandatory access control labels, etc.) are opaque to the applications that use them. The application knows only what types of information are contained in the structure (knows roughly the names of the fields), and not the physical placement or ordering of the structure. Given this, there is no need to standardize on the different structures themselves - only what is contained in them.

The following scheme is used by the different mechanisms to allow applications to manipulate the information contained within the data structures:

1. read in the information from permanent storage to an allocated working storage area.

[2] This specification is now known as P1003.2c.

2. update the information in the working storage area.
3. write the information back to permanent storage.
4. deallocate the working storage area.

For example, an application that would be used to add an entry to an access control list (ACL) would contain interface calls in the following order:

- An interface to allocate a working storage area and to read the ACL from permanent storage into the working storage area,
- An interface(s) to update the ACL entry,
- An interface to write the ACL back out to permanent storage,
- An interface to deallocate working storage.

Once again this scheme provides application developers and programmers the advantage of having to know only the types of information contained in the structures, and not the specifics of the structures themselves. This idea of not being tied to the structures, only the information, certainly allows for application portability.

Each of the following sections that describe the POSIX.6 standard are structured in the following manner:

- **Functionality Overview** - a brief description of the functionality and its intended use,
- **Mechanism Overview** - a brief description of the mechanism used to provide the functionality, and how the mechanism works,
- **Interface Descriptions** - a general look at the specified interfaces.

## 4.3 Audit Trail Generation and Manipulation

### 4.3.1 Audit Trail Functionality

In general, a system's audit trail is a collection of audit records containing data about security relevant events, i.e., attempted violations of the security policy and changes to the security state of the system. When required, applications should be able to generate these audit records. (If the application is an audit analysis tool, the application should also be able to read the audit records.) The audit record should contain attributes such as the time of the event, the status of the event, and the subject(s) and object(s) of the event. In portability terms, a portable application should be able to generate a record (or read the record) containing this information without being concerned about the underlying implementation-dependent record format.

The POSIX.6 audit interfaces allow for this type of portability. The internal structure of the audit record is never seen; it is manipulated only through the read and write interface functions.

The interfaces to the audit trail mechanisms provide portable applications the ability to generate audit records, to select delivery locations for the records, and for some selected applications, the ability to disable and enable the recording of certain events. In most cases these will be privileged operations.

According to the POSIX.6 standard "there are four major functional components of the POSIX.6 audit interface specification:

1. Interfaces for an application to write records to the audit trail and control the auditing of the current process,

2. Interfaces for reading the audit trail and manipulating audit records,

3. The definition of a standard set of events that shall be reportable in implementations,

4. The definition of the contents of audit records."

An audit trail is defined in the POSIX.6 standard as a set of sequential audit records that contain information about security relevant events occurring within the scope of the POSIX.1 standard and any optional POSIX.6 interfaces and objects. This is referred to in the standard as the **system audit trail** and contains records generated by the system or generated by an application. Applications may also write to other audit trails. Interfaces provide the system and applications the ability to write information about security relevant events into the system or other audit trails in the form of audit records. Interfaces provide post-processing applications the capability to read records from the system audit trail or any other audit trail that may exist on the system. The internal format of the audit trail, as well as the audit record format is not defined by this standard. This is consistent with the model of providing interfaces to opaque data structures that was discussed earlier. However an audit record does have a logical structure defined so that post-processing audit applications can call on specific items in the record.

In the context of POSIX.6, audit records are generated in two ways:

1. By a POSIX.6 implementation, to report on the use of its security relevant interfaces. This is known as **system auditing** and the records are known as **system generated records**.

2. By an appropriately privileged application, to report on its own activities. These are known as **application-generated records**.

The use of a logical audit record, as well as a standard set of interfaces to write to the audit trail, and read from the audit trail, allow the system and applications to create, read and manipulate information about security relevant events. This provides conforming implementations and applications a portable mechanism to use in recognizing and reporting on security relevant events.

### 4.3.2 Audit Trail Mechanism Overview

The objects that are created, manipulated, etc. by the audit interfaces are the event specific data within each audit record, and the audit records themselves. To a post-processing application, an audit record logically appears as the following, (as defined by the POSIX.6 standard):

- **header**, provides the version number of the POSIX.6 standard to which the record content conforms; indicates the data format the data record is written in; includes fields for event-time (to be compatible with the time format proposed in POSIX.4), event-type, and event-status. The event-type is the specified result of a POSIX.1 event (interface call) or POSIX.6 event (interface call) that was made. The event-status indicates the result of the event (the event was successful, successful and used appropriate privilege, failed due to access control, failed due to lack of appropriate privilege, etc.).

- **a set of subject attributes**, describes the subject that caused the event to be reported. The user accountable for the event is indicated, as well as possibly the process id, user id, and group id.

- **a set of event specific items**, contains relevant items that are specified to be included for the particular event-type.

- **zero or more sets of object attributes**, describes the objects effected by the event.

For example the call made to the interface chown(), would create an audit record with a defined event-type AUD-AET-chown. The event-specific items would include the pathname for the object, the owner, and the owning group (the parameters used in the chown() call). The object attributes would describe the object effected, e.g., the record could contain the user id and group id of the file before the execution of the chown() call.

A sample of the events that will cause an audit record to be generated include POSIX.1 functions such as changing the owner or permissions of a file, changing directories, creating objects, creating processes (fork() or exec() family), killing processes, creating or deleting links, opening files, using the setuid or setgid features, and others. The POSIX.6 functions that are considered auditable events include opening an audit trail, suspending or resuming auditing for an application, setting access control list information, setting privilege information, setting mandatory access control labels, and setting information labels. Additional events may be defined by an implementation to be auditable events and thus cause audit records to be generated.

According to the POSIX.6 standard, the interfaces specified to support the audit mechanisms can be grouped as follows:

- **Get and release access to an audit trail** - These interfaces are used to select (open) and close an audit trail.

- **Write audit records** - This interface is used after the system or an application constructs the audit record in the required form. A call to this interface will add the header and subject information to the record and append it to the audit trail (in the system-dependent internal form).
- **Read audit Records** - This interface provides the application the next audit record. The interface reads the record into working storage and provides the application a pointer to the record.
- **Control system auditing** - This interface will suspend or resume system auditing of the current process. This is dependent on the audit policy of the system (suspension may not be allowed).
- **Analyzing an audit record** - These interfaces are used to get specific fields from within the record, and are also used to convert the record to human readable text.
- **Storing audit records** - These interfaces allow the system or an application to store a record in user-managed space (perhaps for later post-processing), and conversely allow the system or an application to return the record to system-managed space. This process requires that the record be converted from its internal format to a "byte-copyable" format. These interfaces provide this conversion.

To write an audit record to the audit trail:

1. Open the system audit trail for writing (an audit trail other than the system audit trail can be specified). Opening the audit trail for writing requires appropriate privilege, however opening the trail for reading does not require appropriate privilege. If the call to open is successful, a pointer to the beginning of the audit trail is returned to the calling process. This pointer is then used to access the audit trail.
2. Construct the audit record in user managed space. (The standard assumes that an application or system process would construct the audit record in the managed space of the application or system process).
3. Write the record into the audit trail using the pointer provided.
4. Close the audit record.

To read the audit records in an audit trail:

1. Open the audit trail for reading. The system allocates a buffer area to read the records. A pointer to the beginning of the audit trail is returned. Opening an audit trail for reading does not require privilege.
2. Read the audit record using the pointer returned by the open call. Subsequent read calls return a pointer to the next sequential record.

3. Read the logical pieces of the record (header, subject details, event-specific information, object-specific information) by making calls to the particular area of interest in the record. For example, to access subject details in the record a call is first made to receive a pointer to the header information. Another call uses that pointer to access the subject details of the record. Repeated calls return the data items from the subject details in a predefined order.

4. Close the audit trail.

5. Deallocate the system storage area allocated by the system.

## 4.4 Discretionary Access Control

Discretionary Access Control (DAC) is used to control access by restricting a subject's access to an object. It is generally used to limit a user's access to a file. In this type of access control it is the owner of the file who controls other users' accesses to the file.

Using a DAC mechanism allows users control over access rights to their files. When these rights are managed correctly, only those users specified by the owner may have some combination of read, write, execute, etc. permissions to the file.

### 4.4.1 POSIX.1 Permission Bit Mechanism

The POSIX.1 standard specifies the use of the permission bit mechanism that is currently implemented and used in many POSIX-like systems. This mechanism allows the defined permissions of read, write and execute to be specified for:

1. the file owner,

2. the group of users specified as the "owning group," and

3. all other users (named "other").

This mechanism can be cumbersome to use if permissions need to be specified for a named user who is not the owner (and nearly impossible to specify separate permissions for two users, neither of whom is the owner). It is also not possible to provide specific permissions for different named groups of users. These limitations pointed to the need to provide a Discretionary Access Control mechanism that can provide the granularity of specifying individual users and named groups. The POSIX.6 standard specifies an access control list mechanism to provide this functionality.

### 4.4.2 Access Control Lists

An access control list is an object that is associated with a file and contains entries specifying the access that individual users or groups of users have to the file. Access control lists provide a straightforward way of granting or denying access for a specified user or groups of users. Without the use of access control lists (using the permission bit mechanism only), granting access to the granularity of a single user (who is not the owner of the file) can be cumbersome. The following is a simplified example of an access control list:

| | |
|---|---|
| OWNER: | rwx |
| KAK: | rw |
| LJC: | r |
| JRC: | – – – |
| GROUP1: | rwx |
| GROUP2: | – w – |
| GROUP3: | – – – |
| OTHER: | – – – |

In this example the granting of read, write and execute permission is apparent. User "JRC" and group "GROUP3" are explicitly denied access to the file.

To provide an ACL capability, the POSIX.6 standard specifies:

1. the definition and use of ACLs.
2. the definition of initial access permissions on file creation.
3. the access check algorithm, and (4) the utilities needed to manipulate the ACLs.

The POSIX.6 standard specifies that a POSIX.1 file is the only object that has an ACL associated with it. The POSIX.6 standard does not specify the actual implementation of ACLs on a system, nor does it specify the internal representation of the ACL. Ordering of the entries within the ACL is also not specified, however the internal order does not effect the specified order of the access check algorithm. The composition of an ACL entry is specified by the POSIX.6 standard as follows:

- **Tag type**, specifies that the ACL entry is one of the following: the file owner, the owning group, a specific named user, a specific named group, or other (meaning all other users).
- **Qualifier field**, describes the specific instance of the tag type. For specific named users and specific named groups, the qualifier field contains the userid and groupid, respectively. Qualifier fields for the owner entry and owning group entries are not relevant because this information is specified elsewhere.
- **Set of permissions**, specifies the access rights for the entry.

The POSIX.6 standard specifies that at a minimum read, write, and execute/search permissions must be supported.

The POSIX.6 defined access control list has 3 mandatory entries: an owner entry (called the file owner class), an owner group entry (called the file group class), and a world entry.

This allows the three entries of the permission bit mechanism (owner, group, and other) to also be considered an ACL, and hence, compatible with the POSIX.6 specified ACL interfaces. Calls made to modify these ACL entries will also modify the corresponding file permission bits. Likewise, calls made to modify the file permission bits will also modify the corresponding ACL entries. This is intended to support backward compatibility with the large pool of existing applications that use the interfaces to the file permission bit mechanism.

### 4.4.3 Discretionary Access Algorithm

A process may request to read a file, write to a file, or execute/search a file. To determine this access, the POSIX.6 defined algorithm is applied to the ACL of the file. In general terms, the access check is performed on the ACL entries in the following order:

1. as the file owner.

2. as a named user.

3. as belonging to the owning group, together with any named groups.

4. as belonging in any named groups.

5. as other.

When a match on one of these is made, the ACL is no longer searched, and the granted or denied permissions are in effect. For example, if a user is specified as a named user, and all permissions in the entry are set to deny access to that user, the user is denied access. The groups the user may belong to are not checked to see if the user may have access through the groups' permissions. The algorithm (somewhat simplified here) is as follows:

- If the user requesting access is the file owner, and the requested mode is granted by the ACL entry, then access is granted: else access is denied.

- If the user is a named user in the ACL, and the requested mode is granted by the ACL entry, then access is granted: else access is denied.

- If the user is in the owning group of the file, or is a member of any named groups, and the requested access mode is granted by the ACL entry of the owning group or the ACL entry of any of these named groups, then access is granted: else access is denied.

- If the user is a member of any of the named groups, and the requested access mode is granted by the ACL entry of any of these named groups, then access is granted: else access is denied.

- If the requested access mode is granted by the "other" entry, then access is granted: else access is denied.

### 4.4.4 Discretionary Access Control Interfaces

To read the ACL of a file, a process must have read access to the file's attributes (or possess appropriate privilege). To write (update) the ACL of a file, the process must have write access to the file's attributes and be the file owner (or possess appropriate privilege).

The POSIX.6 interfaces that are specified to implement the access control list mechanism allow a file owner (or a user with appropriate privilege) to create and manipulate an access control list associated with that file. The interfaces for manipulating ACLs and ACL entries can be grouped as follows:

1. Get/set/manipulate ACL entries - includes interfaces to create new entries, copy entries from one ACL to another, and delete entries,

2. Get/set/manipulate ACL entry elements - includes interfaces to add (modify) and delete an ACL entry's permissions or other parts of the entry.

3. Read/write/validate an ACL - includes interfaces that read the whole ACL (the ACL is copied into allocated working space), write the whole ACL (writes the ACL back to permanent storage), and validate the whole ACL (checks for mandatory entries and duplicate entries, as well as sorts the ACL).

4. Translate an ACL into different formats - includes interfaces that allow ACLs to be copied from a system dependent, internal format to a format that can be copied into user managed space, or into a structured text representation.

With these interfaces, portable applications can determine a subject's access to an object, can create new objects and associate an ACL with the object, can manipulate the ACL of an object, and in general use the access information provided by the ACL in a manner that will be consistent across all POSIX.6 compliant systems (that implement the ACL option).

### 4.4.5 Application Considerations

The POSIX.6 standard specifies interfaces and commands for the permission bit mechanism, and there exists a large pool of portable applications that use these interfaces and commands. This implies that backward compatibility with applications that use the permission bit mechanism is necessary, even when the systems using these applications implement the access control list mechanism.

The two DAC mechanisms (the ACL mechanism and the permission bit mechanism) may exist on the same system, and still be POSIX.6 DAC compliant. Great effort was made to ensure that these two mechanisms, if forced to, work together. When possible, interfaces normally used for the permission bit mechanism (i.e., chmod(), and stat()) will work with the access control list, and the interfaces intended for the access control list will work with the permission bit mechanism. However, if the result of an interface has the potential to grant more access than intended, the call will most likely fail. The results of crossing calls may not produce the expected result, but will never be less restrictive than intended.

## 4.5 Privilege

The purpose of a privilege mechanism is to provide a means of granting specific users or processes the ability to perform security-relevant actions for as limited a time and under as restrictive a set of conditions as possible, while still permitting tasks properly authorized by the system administrator. For the administrative task of performing a system backup to be done correctly, all the files of the system must be readable. However one would not consider changing all the access control information for all the files on the system to be readable to accommodate performing backups. The solution would be to create a privilege that would allow a read override of the access control information allowing all the files on the system to be read for the backup procedure. This privilege would then only be used for specified tasks such as system backups. This exemplifies the basic security principle of least privilege.

### 4.5.1 Super-user and Appropriate Privilege

Most UNIX users would expect to find the super-user privilege mechanism to be the standardized POSIX privilege mechanism, but it is not. A goal in supporting privilege was that the base POSIX standard allow for the implementation of a mechanism that supports the least privilege concept described above. The super-user mechanism does not support this. It supports a monolithic "all or nothing" approach to privilege. The only user with any privilege is the super-user (also known as "root" with a UID of 0), and this user has all privileges, all of the time. This clearly does not meet the goal of supporting least privilege, and thus does not exist explicitly in the POSIX.1 standard. Actually no privilege mechanism exists in the standard per se. Only the concept of "appropriate privilege" exists to indicate those services (using the POSIX interfaces) that require privilege. This allows any privilege mechanism to be implemented - including the super-user privilege. However it is the intent that the least privilege mechanism supported by the POSIX.6 standard be used when there is a requirement for privilege, and not the "all or nothing" super-user approach, which does not support least privilege.

The features provided by the POSIX.6 standard, with regard to privilege include: the granularity of privilege, the time bounding of privilege, and privilege inheritance. A privilege mechanism that supports granularity of privilege will allow a process to override only those security-relevant functions that are needed to perform the task. For example a backup program only needs to override read restrictions, and not the write or execute restriction on files. The time bounding of privilege is related in that privileges required by a application or system process can be enabled and disabled as the application or system process needs them. Privilege inheritance allows a process image to request that all, some, or none of its privileges get passed on to the next process image. For example programs that execute other utilities need not pass on any privileges if the utility does not require them.

### 4.5.2 Privileges and Interfaces Requiring Privilege

Under the POSIX.6 privilege mechanism, the granting of privilege is based on the combination of privilege attributes belonging to a process (process privilege attributes) and privilege attributes belonging to a file (file privilege attributes). This allows the mechanism to not be based solely on the user: privileges associated with files are also taken into consideration. The POSIX.6 standard does not preclude that a single user be granted all privileges all of the time (the super-user concept), although this absolute granting of privilege is strongly discouraged from being practiced.

The POSIX.1 interfaces that are covered by the POSIX.6 privilege policies (meaning that appropriate privilege is required) include:

- changing the permission of an object,
- changing the owner of an object,
- creating an object,
- creating a new process (exec()),
- killing a process,
- linking or unlinking an object,
- opening an object,
- using a pipe,
- renaming a file,
- removing a directory,
- using setuid/setgid functions,
- setting the umask (default permissions),
- getting the attribute information of an object.

The POSIX.6 interfaces that are covered by the POSIX.6 privilege policies include:

- reading from or writing to an access control list,
- opening an audit trail,
- suspending or resuming the auditing of an application,
- reading from or writing to an information label,
- reading from or writing to a mandatory access control label,

- reading from or writing to the privilege state of a file.

The set of privileges that are defined by the POSIX.6 standard are somewhat analogous to the functions listed above. For example, opening a file (using the open() interface) requires that the user either be the file owner, or not be the file owner but possess appropriate privilege. Possessing appropriate privilege would mean that the user's process has the priv_fowner privilege. (The priv_fowner privilege allows a process to perform all the functions that file owners have over their files.)

### 4.5.3 Privilege Determination and Privilege Inheritance

The set of privileges that are associated with a process that is executing a file may be revoked, inherited, or absolutely granted. This is all dependent of the value(s) of the file privilege state of the file, and the process privilege state of the previous process image. The process privilege state of a file is defined by the set of process privilege flags associated with a process. The process privilege flags defined by the POSIX.6 standard are permitted, effective, and inheritable. These flags apply to each privilege separately. (That is, a privilege may have some combination of these flags associated with it.) A process can exercise a particular privilege only when the privilege's **effective** flag is set. This flag is the only flag evaluated when determining if a process has appropriate privilege.

A process shall be able to set all the process flags for a particular privilege if the **permitted** flag for that privilege is set. This flag is used to determine whether the effective flag for the privilege will be set, and hence the privilege exercised.

A privilege can be passed on to the next process image only if the **inheritable** flag is set. Whether the inheritance is allowed depends on the file privilege state defined below.

File privilege flags are associated with files. A set of these is applied to each privilege. The POSIX.6 standard defines two privilege flags: allowed and forced. The **allowed** flag permits the privilege to be passed to the next process image depending on the process flag for the previous process. When the allowed flag is set, and the inheritable flag is set for the previous process, then the next process image will have the permitted flag set for that privilege.

The **forced** flag of the file privlege flags allows the privilege to be passed on to the next process image regardless of the previous process image flags or the allowed flag. The process privilege flag for the particular privilege in the new process image automatically becomes permitted.

The last category of defined privilege information is the file privilege attributes. The POSIX.6 standard defines these as values associated with a file that apply to all the privileges defined. There is a single file privilege attribute defined: set_effective. This flag is used by the exec() function to determine which privileges associated with the new process image will be set to effective (and thus possess appropriate privilege).

When a new process is created by the fork() call, the privilege state of the new process is the same as the previous process. When a new process is created by exec(), the following algorithm is used to determine the privilege state of the new process image. This algorithm is applied to each privilege that has its permitted flag set in the current process image.

- If the forced flag is set, then the permitted flag is set in the new process image.
- Otherwise, if the allowed flag is set, and the inheritable flag is set in the current process image, then the permitted flag is set in the new process image.
- Otherwise, if the neither the allowed or forced flag is set, then the permitted flag in the new process image is not set.

The algorithm described above provides the new process image with a set of privileges, each with their permitted flags set. This means that these privileges have the potential to become effective, and thus can be exercised. The factor that determines if these privileges become effective or not is the file privilege attribute: set_effective. If the set-effective attribute of the process file is set, then the effective flag for each of these privileges is set. A process can then exercise appropriate privilege when the privileges that are effective are called for.

Using the mechanism described above, privileges can be revoked (if the inheritable flag is not set), inherited (if the inherited flag and the allowed flag are set), and forced (if the forced flag is set). This capability enables the features described earlier (granularity of privilege, time bounding of privilege, and inheritance of privilege), to be implemented and used on a system. The POSIX.6 defined interfaces allow applications to make calls to use the mechanism to enforce the least privilege principle.

The interfaces specified to obtain, manipulate and set the privileges of a process privilege state or a file privilege state follow the model of using opaque data objects. First a call is made to obtain (read) the privilege state of a file or process. If necessary, working storage is used, the privilege state is read into working storage, and a pointer to the object in working storage is returned. "Get" and "set" interfaces are then used to manipulate the specifics of the privilege state. When finished, the privilege state that is to be associated with a process or file is written to the process privilege state or the file privilege state, respectively.

## 4.6 Mandatory Access Control

The need for a mandatory access control (MAC) mechanism arises when the security policy of a system dictates that:

1. protection decisions must not be decided by the object owner.
2. the system must enforce the protection decisions (i.e., the system enforces the security policy over the wishes or intentions of the object owner).

The POSIX.6 standard provides support for a mandatory access control policy by providing a labeling mechanism and a set of interfaces that can be used to determine access based on the MAC policy.

### 4.6.1 Determining MAC Access

The functionality provided by the interfaces to support MAC is used to determine the access of **objects** by **subjects**. The POSIX.6 standard defines a **subject** to be an active entity that can cause information to flow between controlled objects. The POSIX.6 standard further specifies that since processes are the only such interface-visible element of both the POSIX.1 and POSIX.6 standards, **processes are the only subjects** treated in POSIX.6 MAC. **Objects** are defined by POSIX.6 as the interface-visible data containers, i.e., entities that receive or contain data to which MAC is applied. POSIX.6 specifies that **objects are files** (this includes regular files, directories, FIFO-special files, and unnamed pipes), **and processes** (in cases where a process is the target of some request by another process). POSIX.6 also specifies that each subject and object shall have a MAC label associated with it at all times.

The POSIX.6 standard does not define a mandatory access control policy per se, but does define the restrictions for access based upon the comparison of the MAC label associated with the subject and the MAC label associated with the object. The first general restriction states that unprivileged processes (subjects) cannot cause information labeled at some MAC label (L1) to become accessible to processes at MAC label (L2) unless L2 dominates L1 (see Section 4.6.2 for the definition of "dominates"). This restriction is further defined with regard to accessing files and other processes. The restrictions placed on file manipulation (reading, writing, creating, etc.) are those that are generally accepted when implementing a MAC policy:

1. to read a file, the label of the process must dominate the label of the file.

2. to write to a file, the label of the process must be dominated by the label of the file (The POSIX.6 standard specifies that dominance equals equivalence - if the labels are equal, then each is considered to be dominant to the other).

For example, a user who is running a process at Secret should not be allowed to read a file with a label of Top Secret. Conversely, a user who is running a process with a label of Secret should not be allowed to write to a file with a label of Confidential.

The POSIX.6 restriction for assigning labels to newly created files is that the new file must have a label that is dominant to the label of the subject, although the POSIX.6 interfaces only allow the label to be equal to that of the process creating the new object. This restriction forces implementations to not allow processes to create files at a "lower" label. For example, a process with a label of Top Secret should not be allowed to create a file with a label of Secret. There are analogous restrictions on object access when the object is a process as mentioned above.

Interfaces are provided that allow processes to retrieve, manipulate, compare, set and convert MAC labels. Consistent with the model for using opaque data structures, a label is not manipulated directly but is copied into a working storage area and manipulated there. When the label is no longer requested, the label is written back to its permanent storage area.

### 4.6.2 MAC Labeling Mechanism

The MAC mechanism used with POSIX.6 is a label enforcement mechanism. The access decisions to read (query) objects and write (alter) objects are determined by a general concept of equivalence and dominance between the label of a process (subject) and the label of an object (file, directory, etc.). Defining dominance is left to the conforming implementation, but generally a label "dominates" another label if it is "equal or higher" in some defined structure. For example, in military terms, a label of Top Secret dominates a label of Secret. To read an object, the label of the subject must dominate the label of the object. Reading an object not only includes trying to read the contents of the file, but also trying to read any attribute portion associated with the file, i.e., the access control information, the privilege information, the contents of a directory, directory manipulation, etc. To alter an object, the label of the subject must dominate the label of the object. Manipulating the attribute information of a file is also considered writing to the file, and the MAC-write restrictions are enforced. The POSIX.6 standard does not specify any structure to the security policy that will be the basis for the labels, i.e., it does not require that a lattice model be used.

The MAC label is the item visible at the POSIX interface that is used for mandatory access control decisions. Each subject (process) and each object (files, directories, etc.) shall have a MAC label as an attribute at all times. A physically unique label is not required to be associated with each subject and object, only that a label be logically associated. For example, all the files on one system could share the same label.

The specified interfaces that are used to support the MAC mechanisms are consistent with the model that uses opaque data objects. This means that MAC labels are not manipulated directly, but a copy is placed in a system allocated working storage area, manipulated there, and written back to a permanent area. The interfaces can be grouped into two sets, interfaces that deal with subject and object labels (e.g., reading, writing, duplicating, creating, etc.) and interfaces that deal with label management (testing equivalence/dominance, validating labels, text conversion).

The subject/object interfaces include those that will get (read) and set (write) the label of an object, and get (read) and set (write) the label of the requesting process. To set the MAC label of an object requires appropriate privilege.

The label management interfaces support the following functions:

- **test MAC label relationship** - includes interfaces that will determine dominance between two labels, and equivalence between two labels. It is possible for neither of two labels to dominate the other (i.e., the labels are incomparable),
- **bounding labels** - includes interfaces that will determine the least upper bound of two labels, or the greatest lower bound of two labels,
- **label validity** - include interfaces for determining whether a MAC label is valid. (The definition of "valid" is left to the implementation),
- **text conversion** - includes interfaces that will convert the internal representation of a MAC label into its text representation and vice versa.

## 4.7 Information Labels

There may be instances where security-relevant information (perhaps in a label form) should be associated with subjects and objects and that these labels may not, in general, be used for mandatory access control decisions. Thus, in addition to MAC labeling, the POSIX.6 standard provides a mechanism for data labeling, which makes use of information labels.

Information labels can contain information such as the origin of the object (e.g., that it was created locally, copied from a remote machine, or supplied by a vendor.) a release marking, warning notices pertaining to the object, DAC advisories, project related information, etc. These labels, in general, can be used to support a "data labeling" policy, as opposed to "sensitivity labeling" policies supported by MAC labels.

In addition to the above uses, information labels can be used to trace data flow through a system by using the "float" feature that is unique to these labels. For example, new software that is being introduced into the system for the first time could be labeled "suspect". As the new software is used, the files that become associated with the software would, because of the float feature, become marked in the information label as "suspect" and having this association. If problems occur with the new software, it becomes very easy to see what files have been associated with the software.

### 4.7.1 Information Labeling Mechanism

Information labels have the ability to "float", which is the feature that separates this mechanism from the MAC mechanism. In general terms, an information label is moved "up" or "down" (according to an implementation defined hierarchy) as information is introduced or deleted from the given object. Technically a new label is created for the object that is the combination of the labels of the two parties (subject and object, object to object, or object to subject). The calculation of the new information label is implementation defined. Information labels only apply to the data portion of the file, and not the control portion. Hence, floating occurs only when the data portion of the file is effected.

Information labels used in conjunction with MAC labels can provide useful information that MAC labels alone cannot provide. The two types of labels can use the same label levels. For example, the general MAC restriction of labeling a new object with the same label as the subject (MacLabelA), gives the new object a label of MacLabelA (regardless of whether the information content is actually MacLabelA sensitive). However the object's information content might actually be much lower that of MacLabelA. Using information labels in this scenario would provide the user with additional guidance about the sensitivity of informational content of the file. The information label of the newly created file would represent that the information is at a label "below" MacLabelA, since it does not actually contain information at the sensitivity level of MacLabelA. The information label in this example only provides additional information about the file. It is still the MAC label that is used in access control decisions.

Similar to the MAC labeling scheme, the information label schemes defines both subjects and objects. An information label subject is the same as a MAC label subject, that is, a process is a subject. Information label objects are defined by POSIX.6 as passive entities that contain or receive data. (Unlike the MAC mechanism definitions, the information labeling mechanisms do not consider processes (that are receiving data) and directories to be objects, and thus are not subject to having an information label associated with them.) The POSIX.6 standards considers regular files, FIFO-special files, (unnamed) pipes, and audit trails to be information label objects. POSIX.6 further specifies that each object that contains data must have associated with it an information label at all times. The POSIX.6 standard places restrictions on the use of the mechanism that are similar to the MAC restrictions. The general restriction is that when unprivileged subjects cause data to flow from a source with information label (Label1) to a destination with information label (Label2), the destination's information label shall be automatically set to the value returned by the "float" function that is specified (i.e., float(Label1, Label2)). This means that when information is moved from one file to another, the resulting information label of the receiving file is a combination of the two files. The "combination" of two labels is not specified by the POSIX.6 standard, but is determined by the implementation. Further restrictions that are specified by POSIX.6 for information labels (that also somewhat mirror the MAC restrictions) include: when an unprivileged process with an information label (ILabel1) writes data to a file with an information label (ILabel2), the information label of the file shall automatically be set to the value returned by the "float" function; when a newly created file is assigned an information label, the information label shall be equivalent to the value returned by the "initial information label" label interface. This value is implementation-defined; however, the label must be valid and it must be consistent with the information label policy of the system.

The restrictions placed on processes (subjects) state that when a process with an information label (ILabel1) reads data from a file, or executes a file with information label (ILabel2), the information label of the process shall automatically be set to the value returned by float(ILabel1, ILabel2). Further, a newly created process shall be assigned the information label of the creating subject (process).

### 4.7.2 Interface Descriptions

The interfaces defined by the POSIX.6 standard to support an information labeling mechanism are similar to those of the MAC supporting interfaces, with exceptions made for the labeling float capability. The interfaces support the model that uses opaque data structures, i.e., the information label is copied into working storage from permanent storage, manipulated there, and then written back out to permanent storage. When a function is used that requires working storage, the system must allocate the storage when the interface that requires the storage is called. There are specific interfaces that can be called to free any working storage that was utilized.

The information label interfaces specified support the following functions:

- **test information label relationship** - includes interfaces that will determine dominance between two labels, and equivalence between two labels.

- **float a label** - includes an interface that will produce a label that is the combination of a label associated with a source, and the label associated with a destination. This new label will then be associated with the destination.

- **label validity** - includes an interface for determining whether an information label is valid. The definition of a valid information label is implementation defined; however, examples include: the label is malformed, the label contains components that are not currently defined on the system, or the label is simply forbidden to be dealt with by the system.

## 4.8 Protection and Control Utilities

Draft Standard P1003.6.2[3], Draft Standard for Information Technology - Portable Operating System Interface (POSIX) Part 2: Shell and Utilities - Amendment: Protection and Control Utilities provides the necessary utilities needed for users to utilize the security mechanisms that are provided by the P1003.6 interfaces. The P1003.6.2 Utility standard also specifies necessary modifications to the execution environment utilities specified in the IEEE P1003.2 Shell and Utilities standard.

The P1003.6.2 Utility Standard specifies utilities for four of the five area addressed in the POSIX.6 interfaces standard. The four areas include:

- Access Control Lists (ACL),

- Fine-grained Privilege,

- Mandatory Access Control (MAC),

- Information Labeling (IL).

There are no utilities specified in the audit area because there is no impact on the syntax or semantics of any POSIX.2 utilities or functions, nor is there any need for user utilities to utilize the audit mechanisms specified in the POSIX.6 interface standard.

### 4.8.1 Access Control Lists

There are two utilities specified to access ACL information: **getacl** and **setacl**. The **getacl** displays permission information of ACL entries contained in the ACL of a specified file. This information includes: the file name, the file owner, the file owning group, the permissions of the file owner (the file owner entry), the permissions of the file owning group (the owning group entry), the permissions of named groups (all named group entries), the permissions of all named users (all named user entries), the permissions of "other" users (the "other"

[3]This specification is now know as P1003.2c.

entry), and the permissions of any other implementation-defined entries. The entries are displayed in the order that they are evaluated for access decisions.

The **setacl** utility changes the discretionary access control information associated with a specified file. The options provided by this utility allow a user to: remove all entries except the three base entries (the permission bit mechanism entries), delete entries that are specified from the command line, delete entries that are specified in a named file, update the entries that are specified from the command line, and update entries that are specified in a named file. An entry in an ACL is considered to match a specified ACL entry if the two have equal tag types (ACL_OWNER _OBJ, ACL_USER, ACL_GROUP, etc.) and have equal qualifiers (i.e., the userids or groupids). When using these utilities, the user must specify the file that has the discretionary file information associated with it.

### 4.8.2 Privilege

There are two utilities specified that provide a user with the capability to view or to manipulate privilege information associated with a file or process. The two utilities are **getpriv** and **setpriv**. The **getpriv** utility provides a user the capability to display the privilege attributes of a targeted file(s) or process(es). The options provided by POSIX.6 for this utility include: declaring that the target is a specific process or that the target is a specific file; displaying all values for all privileges, privilege attributes and flags associated with the target; and displaying only those privileges that have at least one flag set, and only those privilege attributes that are set.

The **setpriv** utility changes the privilege state associated with the specified file(s) or process(es). The options provided by POSIX.6 for this utility include: specifying a partial or complete privilege state that consist of one or more privilege specifications that is to be assigned to each target; specifying that the target is either a specified file or a specified process; and reading a partial or complete privilege state from a specified file that is to be assigned to each target.

### 4.8.3 Mandatory Access Control

There are three utilities specified to support mandatory access control. The three utilities are **getfmac** (allows a user to view the label associated with a file), **getpmac** (allows a user to view the label associated with a process), and **setfmac** (allows a user to set the label associated with a file).

The **getfmac** utility displays the text form of a MAC label associated with a file. The user specifies the file. This utility requires read access for the file of the associated label. The **getpmac** utility displays the text form of a MAC label associated with the current process.

The **setfmac** utility changes the MAC label of each specified file to the specified label. The user specifies the file(s) that are to have associated labels changed.

### 4.8.4 Information Labels

There are three utilities specified to support information labeling. The three utilities are **getfinf** (allows a user to view the information label associated with a file), **getpinf** (allows a user to view the information label associated with a process), and **setfinf** (allows a user to set the information label associated with a file).

The **getfinf** utility displays the text form of an information label associated with a file. The user specifies the file. This utility requires read access for the file of the associated information label. The **getpinf** utility displays the text form of an information label associated with the current process.

The **setfinf** utility changes the information label of each specified file to the specified information label. The user specifies the file(s) that are to have associated information labels changed.

## 4.9 Status and Future Work

The P1003.6 Working Group is currently resolving ballot objections to an IEEE ballot for the POSIX.6 interface and utility standards. The ballot ended February, 1993. After making adjustments to the POSIX.6 standards based on the ballot objections, the standards are projected to be re-balloted in the 1993 summer-fall timeframe.

A few prominent issues have surfaced as a result of the recent ballot. One issue is the inclusion of the set of specific privileges defined in the standard. Some who balloted feel that the privileges specified are not granular enough, and some feel that they are too granular. Some who balloted also feel that there is not enough flexibility for those who may not want to require all the specified privileges in their privilege policies.

Another issue is the inclusion of a masking feature that is used with ACLs. When an application needs to temporarily lock a file, the application can make a call to chmod() with the permissions set to 0,0,0 (meaning no one has access). Many existing applications use this type of file locking. When the lock is no longer required, the permissions are set back to their original state. The mask feature allows the chmod() call with permissions set to 0,0,0 to work with an ACL. This provides both portability and backward compatibility to some extent. However this feature also creates more complexity in the access check algorithm, updating ACLs, and other functions. Currently, this feature is not defined by POSIX.6.

The last issue discussed here concerns the use of multi-level directories. A multi-level directory is one in which files of different label levels exist. The /tmp directory is an example of a possible multi-level directory. Interfaces were specified by POSIX.6 for reading and writing multi-level directories, but have since been removed. Currently, multi-level directories are not part of the POSIX.6 standard. These issues will have to be resolved in order for the standard to become stable.

In 1991, the Distributed Security Study Group was formed to study and determine future areas of security that need to be addressed in a POSIX environment. The group focused on distributed environments, an area that was specifically excluded from the original POSIX.6

work. A framework document was produced that examines existing POSIX efforts and how these efforts relate to each other, and also proposes areas of future standardization efforts. As a result of this framework document, the P1003.6 Working Group is currently turning their attentions to the following proposed areas:

- Administrative Services,
- General Cryptographic Services Interfaces,
- Identification and Authentication,
- Networking Services,
- Portable Formats for ACLs, MAC labels, Information labels, File Privilege States, and Audit Trails,
- Security Liaison Efforts with other Posix Working Groups.

# Chapter 5

# Standard Cryptographic Service Calls

**Shu-jen Chang**

This chapter presents an overview of a set of cryptographic service calls being considered for standardization which allows application programs to request cryptographic functions through a standard interface to a cryptographic facility. The set of calls was recently proposed for consideration as an IEEE POSIX security interface standard. The set of calls include basic cryptographic functions, cryptographic key management functions, and user account management functions. Two cryptosystems providing cryptographic functions are addressed: the secret key cryptosystem, and the public-key cryptosystem. A complete description of each cryptographic service call is given in Appendix B. Those readers who are familiar with cryptography and the secret-key and public-key cryptosystems may skip Sections 5.1 and 5.2 and go directly to Appendix B.

## 5.1 Background

Cryptography has been known for decades to protect sensitive or secret information from unintended personnel while the information was delivered via unsecured channels. By encryption, a message is transformed into a form unreadable by anyone without a secret decryption key. Encryption is the only way known so far that can protect the privacy of information traveling in unsecured networks. Cryptography may also be used to protect the integrity of information by a process called message authentication and verification [FIP85]. The process involves the calculation of a computer checksum based on the message to be sent. The checksum is sent along with the message to a recipient, who may recompute the checksum and verify that the message was not modified in transit.

To meet the increasing demand for information security, many vendors have designed and marketed cryptographic products. Though capabilities vary among these products, most of them provide a common set of cryptographic functions, including message encryption/decryption, message authentication/verification, and key management functions. Currently, two types of cryptosystems prevail: the secret-key cryptosystem and the public-key cryptosystem. Regardless of the technique used, each has some keying information to protect and manage, thus each needs a properly implemented key management system.

To protect the secret information such as users' secret keys from unwanted disclosure, cryptographic functions are frequently implemented in firmware or hardware and protected by tamper-proof casings. This component is frequently referred to as a cryptographic module (CM), which is a set of hardware, firmware or software, or some combination thereof, that implements cryptographic logic and/or processes [FIP94]. The cryptographic module may also be used to generate new keys and to encrypt keys for storage. The FIPS 140-1 [FIP94] details the security requirements for cryptographic modules. It is obvious that without proper protection to the cryptographic module, any security service provided by it is totally useless.

To use these security products, vendors normally provide executable programs that users can load into their host computers and execute. Frequently, high-level procedure calls are also provided to permit customized application programs in the host computer to interface with the CM and request security services through these cryptographic service calls. The cryptographic service calls may be referred to as the cryptographic application program interface (API) by some vendors. It is likely that different vendors may provide different cryptographic APIs for their products, since each product is designed and implemented differently. It is even possible that different cryptographic products from the same vendor may have different sets of APIs. It is felt that a standardized interface for basic cryptographic functions will be very beneficial for application programmers. If all the cryptographic modules manufactured in the future can support this standard cryptographic service interface, an application program written for a particular CM may well be ported to work with a different CM without any modification. The Security Technology Group of the Computer Security Division at NIST has been developing such a generic set of cryptographic service calls to be published as a FIPS guideline and proposed it for an IEEE POSIX security interface standard. The set of cryptographic service calls has sustained several reviews within and outside the Division, however, since it is still a draft document, future modification is possible.

## 5.2 Overview of Secret-Key and Public-Key Cryptography

This Section presents an overview of the cryptographic service calls for the secret-key and public-key cryptosystems. To fully understand these service calls, readers may wish to read [Rus91] to learn more about the two prime cryptographic technologies.

In secret-key cryptography, a secret key is established and shared between two individuals or parties and the same key is used to encrypt or decrypt messages, therefore, it is also referred to as symmetric cryptography. If the two parties are in different physical locations, they must trust a courier, or some transmission system to establish the initial key and trust this third-party not to disclose the secret key they are communicating. The generation, transmission, and storage of keys is called key management. Ensuring that key storage, exchange of new keys and destruction of old keys are performed securely often creates complex key management requirements for secret key cryptography. The ANSI X9.17 Financial Institution Key Management (Wholesale) Standard prescribes a uniform process for the protection and exchange of cryptographic keys for authentication and encryption in the financial community [ANS85].

In a public-key cryptosystem, a user makes use of a pair of keys: a public key and a private key. The public key of a user can be made public without doing any harm to security, while the private key of a user never leaves the possession of its owner, which is increased security over the secret key cryptography [Fah92]. With public key cryptography, no single key is used for both encryption and decryption, thus, it is also referred to as asymmetric cryptography. It is beyond the scope of this document to describe how public-key encryption works, interested readers are referred to [NIS91b] for the details. Since a user's public key is made public, certain control is necessary so that a user's public key cannot be tampered with. The application of public-key cryptography thus requires an authentication framework which binds users' public keys and users' identities. A public-key certificate is a certified proof of such a binding vouched for by a trusted third-party called a Certification Authority (CA). The use of a CA alleviates the responsibility of individual users to verify directly the correctness of other users' public keys. Public key certificates are managed by a certificate management system, the development of which is very complex. Reference [Nec92] gives a detailed discussion of the issues involved for managing public-key certificates.

# Part III

# Human/Computer Interaction Services Security

# Chapter 6

# General Issues

**Robert Bagwill**

A brief discussion of the general security issues related to the human-computer interface follows. The two main security responsibilities of the system are:

- Identifying and authenticating users, programs, and other systems.
- Restricting user, program and other systems' activities to those whom have been authorized.

Identifying users is discussed below. Authenticating users, programs, and systems is discussed in Chapter 10. Restricting user, program, and other systems' activities is discussed in Chapters 4 and 10. And given that the human/computer interface has a physical component, some of the issues of hardware security are discussed.

## 6.1 Identifying Users

Generally, a user's first activity when starting a session with a secured computer system is identifying himself/herself to the system. The most common ways to identify and authenticate users are by the use of physical keys, account names and passwords, and biometric checks.

### 6.1.1 Physical Keys

A physical key is an object whose characteristics are somehow secret, and which is usually somewhat difficult to reproduce. It could be:

- a piece of machined metal that unlocks the computer;

- a *hardware device* that attaches to an I/O channel (e.g., a serial line with an RS-232 connector), which can be interrogated by the system, and which must be present to execute certain programs;

- a *smart card*, which is a credit-card-sized circuit board which contains some form of non-volatile memory, and may even have a CPU.

The risks of physical keys are familiar and obvious:

- keys can be forgotten, broken, lost, borrowed or lent;

- keys can be stolen, or copied (by a determined user);

- keys and compromised locks can be expensive to replace;

- it can be difficult or impossible to automatically or remotely revise authorizations associated with a particular physical key.

- physical keys must be physically managed, i.e., stored, logged, kept secure, etc.

Often a physical key is used with a password or biometric check.

### 6.1.2 Passwords

A password is a sequence of characters which is a shared secret between the user and system. Passwords are usually stored on the system in a user-inaccessible location, or are stored in an encrypted form. Password present a variety of risks:

- passwords can be guessed, shared, written down, or forgotten;

- passwords can be stolen by observation;

- passwords tend not to be changed very often, and if they are, are more readily forgotten;

- passwords in plaintext are passed over the network, or are stored in publicly readable locations on the system.

- encrypted passwords are often publicly readable, making them susceptible to cryptographic analysis.

- short passwords can be found via brute-force methods

These risks led system architects to search for other identification methods, including biometric checks.

### 6.1.3 Biometric Checks

A relatively new method of identification for computer systems is the biometric check. It consists of comparing some readily accessible and reliably unique physical characteristic of a human user against the system's stored values for that characteristic. Some commonly used biometrics are:

- hand proportions
- facial image
- retinal image
- finger prints
- voice print

The advantages of biometrics are that they cannot be lent like a physical key or forgotten like a password. The drawbacks of biometric checks are obvious:

- all the biometric sensors are relatively expensive, in both monetary and computing terms;
- measuring hands requires that the appropriate hand be free, ungloved, and that the user has a measurable hand;
- a facial image scan requires that the user's appearance not change drastically;
- a retinal image scan requires that the user has a measurable retina, that eyeglasses or a contact lens not interfere, and that the user is willing to allow the scanner's laser to scan their eye;
- fingerprint analysis has the same drawbacks as the hand proportion metrics;
- voice print analysis is affected by noise and throat problems, and requires that the user have a measurable voice.

## 6.2 Platforms

Each category of hardware/software platform has its own strengths and weaknesses with regard to its human-computer interface and security.

### 6.2.1 Personal Computers

As organizations begin to install Open Systems operating systems and programs on their personal computers, the limitations of those platforms must be recognized and handled appropriately. Traditionally, most personal computers were designed to be single-user, single-tasking systems. As a result, many of the safeguards one usually associates with multi-user, multi-tasking systems are reduced or absent. Currently, some personal computer operating systems support a limited form of task-switching or cooperative multi-tasking. In general:

- Although some personal computers have keyed locks, the keys are not necessarily unique, so one size fits all.
- Password protection of the machine is absent or not enabled.
- There is no built-in support for sharing a single machine between multiple users.
- Every program has unlimited access to all the hardware, and by extension, all the software. As a result:
  - Any program can modify the hardware and software interrupts and timers.
  - Any program can read or write any area of memory.
  - There is little or no protection against the inadvertent or intentional modification or deletion of files.

Fortunately, most Open Systems operating systems provide the needed safeguards.

### 6.2.2 Workstations

Multi-user, multi-tasking workstations generally have operating system architectures that address the weaknesses associated with personal computers. Most workstation operating systems support passwords, provide an insulating and protective layer of software between the user program and the hardware, and provide memory isolation between user program processes and the operating system.

However, there are other weaknesses that usually are not addressed. Some of the relevant UNIX-derived OS weaknesses are:

- Programs do not run in a "least resource" environment. Any application can potentially monopolize all the resources of a system.
- Programs do not run in a "least privilege" environment. That is, although a user may expect an application to affect only the files the user specified, a program can actually manipulate any resource the user can manipulate.
- Tty's and pseudo-tty's are not handled in a secure manner. A program can open a terminal and wait for another program to open the same terminal.

- Programs share the same directories to create temporary files, which means a program can modify or delete the work files of other programs.

- Programs are built with shared libraries containing references to relative rather than absolute pathnames.

- Programs that access a file without checking if it is a symbolic link can be tricked into accessing a substitute file.

- Setuid programs (i.e., those programs capable of running with a user ID other than the one which is the owner of the executable file and/or capable of changing user IDs while running) are inherently unsafe.

- Network conveniences like hosts.equiv and NFS have many bugs, security holes, and potential management pitfalls. Chapter 9 discusses network security threats and Chapter 10 describes ways to improve security in a network environment.

Most of these problems *are* addressed by the Compartmented Mode Workstation requirements, but are usually not addressed by the normal, commercial workstation platforms.

### 6.2.3 Servers

For the purposes of this discussion, a server is a multi-user, multi-tasking computer system that is intended to provide simultaneous service to multiple users. It often runs what are considered *mission critical* applications. Nowadays, workstations and servers often run the same operating system, so all the shortcomings of the workstation operating systems are shared by the server. In addition, servers are most often accessed via a network, rather than multiple serial connections, so they are more sensitive to the risks of networking.

## 6.3 Hardware Security

All software security depends on hardware security. If the hardware can be stolen or surreptitiously replaced, secure software will not help. When computers filled a room, stolen computers were not a big problem. Now that laptop and palmtop computers are the fastest growing market, physical security is at least as important as software security.

Some of the most common problems are:

- equipment and removable media is stolen or replaced;

- security can be circumvented by changing hardware setup parameters;

- systems can be booted by unauthorized users;

- systems can be booted from unauthorized software;

- boot media can be re-written by unauthorized software, and
- unauthorized software can be executed from removable media.

Some of the safeguards which can be taken are:

- locked doors and secured equipment;
- lockable cases, keyboards, and removable media drives;
- key or password-protected configuration and setup;
- password required to boot;
- password required to mount removable media;
- read-only boot media, and
- storing removable media in secured areas.

Most of these safeguards are required for the DoD's System High and Compartmented Mode Workstations, which are briefly discussed in a later section.

Other problems related to hardware are eavesdropping via electro-magnetic interference (EMI) detection and analysis, and communications interception. EMI detection is out-of-scope for this report. Communications interception is discussed in Chapter 9.

## 6.4 Training

Last, but not least, users need training in the correct use of the system. Untrained users can intentionally or unintentionally subvert security policies through lack of training.

# Chapter 7

# The X Window System

**Robert Bagwill**

## 7.1 Introduction to the X Window System

The X Window System is a network-transparent graphical user interface technology for bitmapped displays [Sch91]. It is a collection of protocol definitions, file formats, documentation and sample software source code in C for server, client, and utility programs. It was designed to be portable between different operating systems and display hardware. It was developed by the Athena Project at MIT, which was supported by IBM and Digital. Its continued development has been taken over by the X Consortium, which is made up of developers, vendors and users of X. The X Window System is copyrighted by MIT and the X Consortium, but the specifications and source code are freely available. As a result, it has been widely accepted by the computer industry.

This near-universal acceptance has advantages and disadvantages from a security perspective. On the one hand, X's wide usage helps make it more robust, and ensures that security concerns are also widely felt, leading to better security solutions. On the other hand, the freely available source and specifications, and the wide publication of X's strengths and weaknesses make "Security through Obscurity" impossible.

What distinguishes X from most other graphical user interfaces is the way its architecture splits the graphics functionality between the application itself, which is known as the X client, and another program, known as the X server (all subsequent references to a client or application should be read as "X client application").

In the database world, there are similar client/server architectures. A database server program, often running on a mainframe computer, listens for and responds to database requests from database client programs, which usually run on workstations of some type. The database server maintains exclusive control of the database, and multiplexes requests and synchronizes database activity.

In the graphics world, the X server program maintains exclusive control of the display and

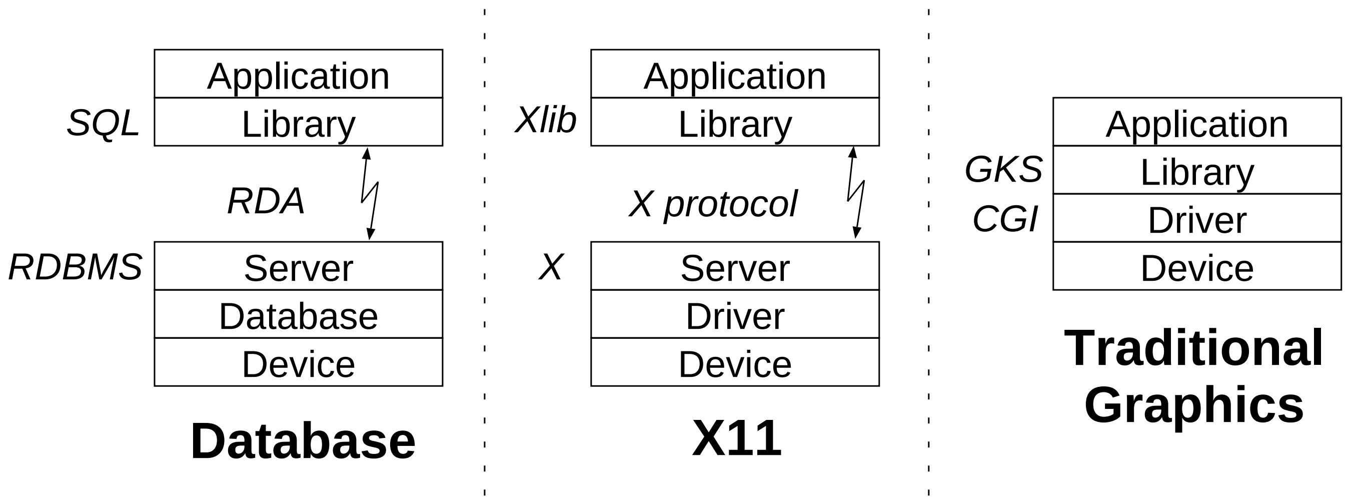



Figure 7.1: Comparison of Architectures.

multiplexes requests from X clients. Just as multiple database clients can share the database via the database server, multiple graphical client applications can share a display via the X server. The graphical client/server protocol is the X protocol. The X protocol can be used over any inter-process communication mechanism that provides a reliable octet-stream.

This is in contrast with the traditional graphics architecture, which consisted of an application which made calls to a graphics library which in turn made calls to a device-dependent graphics driver which controlled the display. If one wanted to port the application to a new machine, operating system or display, one might have to re-write the application *and* graphics library *and* graphics driver. If one used a standard graphics library like GKS, one might avoid rewriting the application and library, if there was a compatible library and graphics driver for the new machine. But the graphics output would still be confined to a single machine (see fig. 7.1).

X provides a common low-level programming interface (known as Xlib), a common protocol (X) and a common device-independent replacement for the graphics driver (the X server). So porting a client to a new machine is relatively easy, and the X server only needs to be ported once for each display architecture.

The X architecture allows a single display to show the output of programs running anywhere on the network. This permits the user to run a CPU-intensive problem on a Cray, run a mail program on a Sun, and run a wordprocessor on a Compaq, and display all the programs on a single screen [Hel90]. Figure 7.2 illustrates this concept. The X11 Server drives the display of the user's workstation. An X application (the X client) using the toolkit can run on a Cray connected to the network, display results in a window on the user's workstation, and accepts input from the user's keyboard and mouse. Likewise, another X client application can run on a Sun connected to the network and display results in another window on the user's screen.

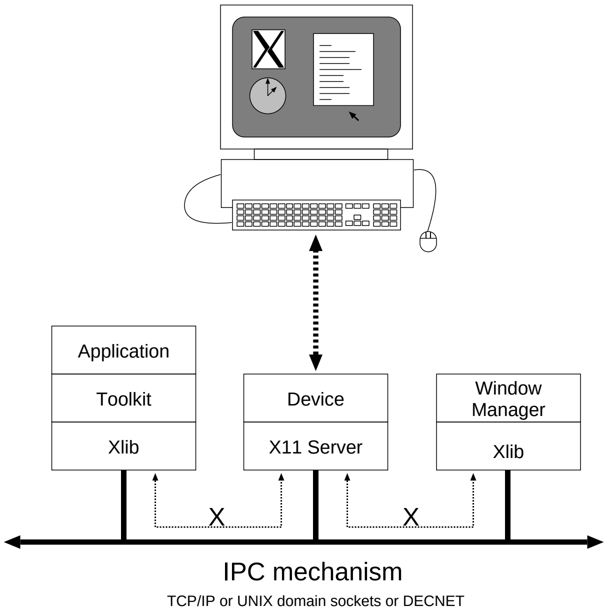



Figure 7.2: X Window System Architecture.

## 7.2 The X Server

Because the X Server allows various X clients to share the resources of a single display, there is a potential for conflict.

### 7.2.1 Events

Server and input activities are relayed to a client as messages called *events*. These events include keystrokes, pointer movements, color changes, etc. When each window of a client is created, a list of the events it is interested in receiving is sent to the server. When some input or server activity occurs, the server checks which windows should be notified. Although a client normally only asks to be notified for events in its own windows and subwindows, a client *can* request to be notified about all events for any window. As a result, a client can "eavesdrop" on the activities of any other client on a given display. In particular, this means that a client can intercept keystrokes associated with logins made from another window on a given display. Such keystrokes include a login password.

### 7.2.2 Properties and Resources

The X server maintains an active database whose contents are called *properties*. A property consists of a name, type, and contents. Applications can use properties for any purpose, but they are most often used to communicate between clients, and between a client and the window manager or session manager. There is a special property list associated with the root window known as the *resource database* that is managed by the *Resource Manager*. A set of Xlib routines allow an application to get, set, and query the resource database. This database may be used by clients to set application defaults, to specify keymaps, or to register the functions to call when a particular window receives a particular event.

An application may read or modify any resource entry. The effects of changing an entry depend on how the application or applications use that resource. For example, an application could set the default foreground and background colors for all applications to black.

### 7.2.3 Fonts

When an X client requests a font, the X server loads it into memory from local storage, from a file system, or from a font server on the network. When the X fonts are retrieved from a file system, the X server has a list of directory paths to search for the file containing the requested font. All clients share access to the available fonts. An X client can change the X server font path. As a result, when another client requests a particular font, the X server may not be able to find it, or may load a different font of the same name from that directory.

### 7.2.4 Other Resources

Other server resources that clients share are pointer bitmaps, backing store, and execution time.

Backing store is extra, off-screen memory that an application can request for a server to use to save obscured areas of the client's windows. Some applications become unusably slow if they cannot use backing store.

Execution time is the time the server spends executing the X requests made by the applications. One application can monopolize the server by flooding it with requests.

### 7.2.5 Extensions to X

X was designed to be extensible. Anyone can invent an extension to the protocol for their own use. Most people who develop X extensions eventually submit them to the X Consortium to be added to the distribution. Two of the more popular extensions are Display PostScript and PEX (PHIGS Extensions to X). Other extensions to support multiple input devices, video, and compressed images have been suggested or added.

Problems:

- each extension must be analyzed for new security problems;
- if a client requires the use of an extension that the target server does not support, the application will fail;
- each extension increases the size of the server and client libraries, which can make clients less likely to be responsive in a timely manner;

- supporting an extension typically creates more work for the server, which can decrease availability;
- less popular extensions are likely to be less robust;
- the run-time requirements of an extension may limit its use to more expensive platforms, or may decrease its robustness on low-end platforms.

## 7.3 Inter-Client Communication Conventions Manual

Because of the characteristics of X described in the previous section, there is a notion of a well-behaved application. Some of the recommended behaviors are documented in the Inter-Client Communication Conventions Manual [Ros91].

> It was an explicit design goal of X Version 11 to specify mechanism, not policy. As a result, a client that converses with the server using the protocol defined by the *X Window System Protocol, Version 11* may operate correctly in isolation but may not coexist properly with others sharing the same server.
>
> Being a good citizen in the X Version 11 world involves adhering to conventions that govern inter-client communications in the following areas:
>
> - Selection mechanism
> - Cut buffers
> - Window manager
> - Session manager
> - Manipulation of shared resources

One of the primary mechanisms to support the conventions is the use of properties.

### 7.3.1 Selections

Selections are the primary mechanism used for the exchange of information between clients. Selections use the property mechanism to exchange data through the server and are global to the server.

### 7.3.2 Cut Buffers

Cut buffers, like selections, allow an application to store data in the server which can be retrieved by any application. All the cut buffers are readable and writable by all applications. A client can interfere with the use of cut buffers by other applications.

### 7.3.3 Window Manager

One of the advantages of X is that multiple applications can share the same screen space. Given that screen space is a limited resource, there needs to be mechanisms for sharing it.

### 7.3.4 Session Manager

The session manager is a special client which will manage of collection of clients. It is usually used to start and stop a collection of clients specified by the user. For example, on login, the session manager might start a mail reader, calendar, and editor. Or it might restart the collection of clients which were running when the user last logged out. To be managed by the session manager, clients must provide it with information necessary to restart them. These properties include the text string which can be executed to start or restart the client, the name of the system running the client (as opposed to the system which is running the X server), and the state of the client (normal, iconic, or withdrawn). The client must also be ready to respond to messages from the session manager, such as a message to save their internal state before termination, or to delete a window.

A client can specify any value for the properties it shares with the session manager. A client could specify a command which would delete all the user's files on restart, or it could specify that it be restarted on a machine other than the one it was started on.

### 7.3.5 Manipulation of Shared Resources

#### Grabs

To provide more control over interaction with the user, a client can request exclusive access to the server. This is known as "grabbing." A client can grab the pointer, the keyboard, or the entire server. This allows a malicious or misbehaving application to prevent other clients from receiving the events they need to operate correctly.

#### Color

X supports several color models, including monochrome, greyscale, pseudocolor and true color. With monochrome and true color, there is a one-to-one mapping between the colors available to an application and the colors supported by the display hardware. So if the display hardware supports 16 million different colors simultaneously, an application can use as many of the colors as it needs. Pseudocolor supports the mapping of a limited number of application colors to a large number display colors. When a vendor states that a particular hardware/software combination supports 256 different simultaneous colors out of a palette of 16 million, that is describing the use of pseudocolor. The table that maps each of those 256 application colors to an actual display color is called a colormap.

An application may share a colormap with other applications, or may allocate a private colormap. If the shared colormap is writable, any application may change the colors in the colormap. This allows a malicious or misbehaving application to deny access to the shared colormap to other applications, or to change the on-screen appearance of other applications by modifying the shared colormap.

#### Keyboard

X was designed to be portable between machines with different keyboard hardware. It provides a way to map physical or logical keys to characters of a national language. An application can modify the keyboard mapping, which can cause aberrant behavior in other applications.

## 7.4 Platforms

The X Window System runs on a variety of platforms. On personal computers and X terminals, the X server may be the only program running on the system. On workstations, it is usually one of a number of applications which are running simultaneously.

### 7.4.1 Networking

X has been used over TCP/IP, DECnet, and LAT over ethernet. It has also been over SLIP, PPP, and other proprietary protocols over serial lines. The risks associated with TCP/IP are discussed in Chapter 9. There has been a mapping of X to the OSI protocol stack and services. As yet, there are no commercially-available implementations of X as an OSI Application Layer protocol. When there are, one would expect X to use the OSI security services, which are under development.

#### Serial

Because X can run over any reliable octet-stream interface, X can be used over a serial line. X may be used where one or more of the following is true:

- bandwidth requirements are low
- installing higher bandwidth cabling is infeasible
- the user is mobile or at a remote site
- public voice-grade lines are the only available networking option

Problems:

- data transfer is over public lines or over interceptable cellular channels
- encrypting modems are not widely used
- serial cables are easy to tap
- service quality over public networks degrades quickly under load

### 7.4.2 Personal Computers

A personal computer, running a classic single-user single-tasking operating system, usually runs an X server as a dedicated application. When the X server is running, no other applications run. All the X clients are running on a server system elsewhere. The personal computer is connected to the server via a serial line or some type of network connection such as ethernet. If it is talking over ethernet, the personal computer identifies itself to the other system using its IP address and its ethernet address. An unauthorized user can change the IP address, and sometimes the ethernet address, to masquerade as another system.

### 7.4.3 X Terminals

X terminals are special-purpose computers with a display, one or more input devices, and some kind of communications interface. The X terminal runs a copy of the X server, and usually all X clients are running on other hosts. Most X terminals have thin, thick, or twisted-pair ethernet communications interfaces, and support TCP/IP. Several vendors sell X terminals with serial interfaces and an X server which understands their proprietary serial version of the X protocol. Others sell X terminals which use Digital Equipment Corporations LAT protocol. X terminals are attractive because they are less expensive than workstations and they require less management.

Some insecurities which are somewhat peculiar to X terminals (although diskless X workstations may also be vulnerable):

#### Configuration Parameters

Most X terminals allow any user to change the configuration parameters, including such things as the TCP/IP address, servers addresses, display manager behavior, and local clients (if any).

A malicious or mistaken user can:

- change or erase any parameters,
- use an X terminal to masquerade as another host, or
- change the parameters to cause the user to login to an untrusted host.

#### Reverse Address Resolution Protocol

Rather than having the TCP/IP address stored in non-volatile ROM, some X terminals broadcast a RARP packet, asking that a server system tell them what their address should be. The first host to respond can set the terminal's address to anything.

This allows a malicious system manager to disable X terminals, or to cause the user to execute a fake login program, revealing their password.

### Trivial File Transfer Protocol

Many X terminals download the X server image from another host, rather than having the server image stored locally in non-volatile ROM. Any host that masquerades as the TFTP host can download any code to the X terminal.

Also, the TFTP protocol does not do any authentication of requests, so that a malicious client can download files that it should not have access to, or can cause denial-of-service by flooding the host with TFTP requests.

### Fonts

Most X terminals have some application fonts resident and download the others from other hosts. Again, any host masquerading as the TFTP host can download any font. Erroneous fonts can cause application errors, or crucial symbols could be replaced. For example, the "exit without saving" symbol could be replaced with a "save and exit symbol."

X provides the capability for implementing different access control mechanisms. Release 5 includes four mechanisms[Sch91]:

- Host Access Simple host-based access control.
- MIT-MAGIC-COOKIE-1 Shared plain-text "cookies."
- XDM-AUTHORIZATION-1 Secure DES based private-keys.
- SUN-DES-1 Based on Sun's Secure RPC system.

## 7.4.4 Xhost

The xhost program is used to add and delete hosts to the list of machines that are allowed to make connections to the X server. This list of hosts permits access to any user on one of the named hosts. Because this mechanism is unable to permit or deny access to specific users, it provides only a rudimentary form of privacy control and security. It is only sufficient for a single user environment, although it does limit the worst abuses. Environments which require more sophisticated measures should use the hooks in the protocol for passing authentication data to the server. [Sch91]

## 7.4.5 Xdm

As the Xdm manual page explains:

> Xdm manages a collection of X displays, both local and possibly remote – the emergence of X terminals guided the design of several parts of this system, along with the development of the X Consortium standard XDMCP (the X Display Manager Control Protocol). It is designed to provide services similar

> to that provided by init, getty and login on character terminals: prompting for login/password, authenticating the user and running a "session."
>
> A "session" is defined as the lifetime of a particular process; in the traditional character-based terminal world, it is the user's login shell process. In the xdm context, it is an arbitrary session manager. This is because in a windowing environment, a user's login shell process would not necessarily have any terminal-like interface with which to connect.

Xdm can make use of the MIT-MAGIC-COOKIE-1 authorization, detailed below.

### 7.4.6 MIT-MAGIC-COOKIE

The MIT-MAGIC-COOKIE protocol allows Xdm to create a hard-to-guess token that is only readable by the user account which successfully logged in via Xdm. It uses the Unix file system access control to protect the token. The user can copy this token to the user's home directories on other systems to allow clients on those hosts to connect to the X server. [Sch91]

> When using MIT-MAGIC-COOKIE-1, the client sends a 128 bit "cookie" along with the connection setup information. If the cookie presented by the client matches one that the X server has, the connection is allowed access. The cookie is chosen so that it is hard to guess; *xdm* generates such cookies automatically when this form of access control is used. The user's copy of the cookie is usually stored in the *.Xauthority* file in the home directory, although the environment variable *XAUTHORITY* can be used to specify an alternate location. *Xdm* automatically passes a cookie to the server for each new login session, and stores the cookie in the user file at login.
>
> The cookie is transmitted on the network without encryption, so there is nothing to prevent a network snooper from obtaining the data and using it to gain access to the X server. This system is useful in an environment where many users are running applications on the same machine and want to avoid interference from each other, with the caveat that this control is only as good as the access control to the physical network. In environments where network-level snooping is difficult, this system can work reasonably well.

### 7.4.7 SUN-DES-1 and Kerberos

SUN-DES-1 uses the secure RPC facilities of SunOS to authenticate clients to the server. A public key database is maintained on a master machine on the network. The database contains the user's public key and the user's secret key which has been encrypted with the user's login password. Using this system, the X server can securely discover the actual user name of the requesting process. It involves encrypting data with the X server's public key, and so the identity of the user who started the X server is needed for this; this identity is stored in the *.Xauthority* file.

Kerberos is an authentication protocol developed to support MIT's Athena project. It uses an authentication server which exchanges authentication tokens between two potential clients.

Both protocols are discussed in more detail in the Chapter 9.

## 7.5 Compartmented Mode Workstations

The DoD Intelligence Information Systems (DODIIS) program needed to describe systems which would address the same security issues which were described in the General Issues chapter, as well as the more stringent requirements of the DoD security policies [Woo87]. This includes the capability of displaying data, which is classified at different levels, within different windows at the same time. They created a series of documents that described the workstation requirements, guidelines for implementing the requirements, and the criteria for evaluating an implementation.

The requirements for Compartmented Mode Workstations (CMW) are structured in terms of:

- Access Control and Labels;
- Accountability;
- Operational Assurance;
- Life Cycle Assurance;
- Documentation;
- Environmental Protection; and
- Administrative Procedures.

### 7.5.1 Access Control and Labels

Access control is the ability to selectively allow other users access to information. UNIX-style access controls support user, group, and world read/write/execute permissions. The Access Control Lists (ACL's) specified by C2 implementations and the CMW requirements allow finer-grained control. A single user may be granted access to a file, or may be excluded from a group that has access to a file.

Labels are security-related information which is associated with objects like windows, processes, files, or devices. The ability to associate security labels with system objects is also under security control. CMWs can utilize two forms of security labels: mandatory access and information labels. Mandatory labels are static for the particular object. Information labels may change as data is put into the object.

### 7.5.2 Accountability

Accountability covers:

- user identification and authentication;
- identification of user terminals;
- trusted path between the user and the system; and
- auditing.

User identification and authentication are as described in the previous chapter.

Identification of user terminals allows the system to know which physical terminal a user is using.

A trusted path is a secure means of communication between the user and the system. For example, when a user types in their account name and password, the user wants to be sure that it is the system that the user is talking to, not a malicious program that someone else has left running on the terminal.

A particular implementation may require that the user press a "break" key that reliably terminates any previous session, or the user may need to reboot the personal computer or X terminal.

Auditing logs any security-related event to a secure logfile. Typical events which are logged are logins, logouts, creating or deleting files, modifying the access control associated with a file, and so on.

### 7.5.3 Operation Assurance

Operation assurance covers:

- system architecture
- system integrity
- trusted facility management
- trusted recovery

System architecture addresses the problems described in Section 6.2.

System integrity addresses hardware self-tests and software checks that the appropriate version of the hardware and software is being used.

Trusted recovery addresses the need to be able to recover the system after a hardware or software failure which compromises the protection of the system.

### 7.5.4 Life-cycle Assurance

Life-cycle Assurance covers:

- security testing
- design specification and verification
- configuration management
- trusted distribution

Life-cycle assurance addresses the management functions which are necessary to preserve and insure the integrity of the system.

### 7.5.5 CMW and X

The CMW criteria describe what the conforming CM workstation must do, but it is up to vendors of CMWs to implement those features. It is evident that the typical commercial X workstation does not include those features, and that a considerable amount of effort is required to upgrade the standard X distribution to support them.

# Part IV

# Data Management Services Security

# Chapter 8

# SQL

**John Barkley**

## 8.1 Security with SQL

Database applications are pervasive in any organization. In these database applications, databases organized according to the relational model are among the most widely used. The database language SQL [ANS92] provides a standard means of accessing data organized according to the relational model. In this section, security features provided by SQL are described. The use of SQL in a network environment is specified by the RDA standard ([ISO90b] and [ISO90c]). This section also discusses the security considerations of using SQL in a network environment.

The term SQL is often used to refer to different specifications or implementations. In the formal standards area, SQL'89 refers to ANSI X3.135-1989 [ANS89] (FIPS 127-1 [FIP90]); SQL'92 refers to ANSI X3.135-1992 [ANS92] (FIPS 127-2 [FIP93a]); and SQL3 refers to ANSI X3-135-199x [ISO92]which is expected to be approved by ANSI within a couple of years. In addition, the term SQL is often used to refer to a SQL implementation which includes vendor enhancements. In this section, the term "SQL" always refers to one of the ANSI standards. If a capability described is contained within each of SQL'89, SQL'92, and SQL3, then the term "SQL" is used. If a capability is not contained within each of SQL'89, SQL'92, and SQL3, then the versions of the standard which support the capability are specifically named.

SQL'89 provides basic facilities for creating and manipulating databases based on a relational model. These facilities include:

- Schema Definition: the ability to declare the structures and access privileges of a database.

- Data Manipulation: the ability to populate a database and access that data.

- Transaction Management: the ability to define and manage SQL transactions.

SQL'92 provides additional capabilities including schema manipulation and the ability (called dynamic SQL) to dynamically build and execute SQL statements. SQL'92 also provides the means to apply SQL to a network environment by adding the capabilities for connection and session management. SQL3 will provide the ability to define, create, and manipulate more general objects in addition to tables.

### 8.1.1 Using SQL

SQL may be used by direct invocation or in conjunction with a programming language. When invoking SQL directly, the methods used to accomplish the invocation and return results are implementation-defined. Not all SQL statements may be invoked directly. Those statements that can be invoked directly are referred to as *direct* SQL statements.

SQL does not have control statements, i.e., statements which control program flow, such as, branch-on-condition statements and looping statements. Consequently, SQL is commonly used with other languages that do have control statements. SQL'92 specifies how SQL may be used in conjunction with the standard programming languages Ada, C, COBOL, Fortran, MUMPS, Pascal, and PL/I. SQL is used with a programming language in two ways: by means of modules or by means of embedding.

#### Module Language

A module language defined within SQL is used to create modules containing SQL statements. A module contains a set of procedure definitions where each procedure consists of parameter declarations and a single SQL statement. A procedure may be invoked by the procedure *call* statement of the programming language. The module, which is usually processed into some intermediate form, is generally associated with the calling program either when the program is linked or when the program begins execution.

#### Embedded SQL

SQL statements may also be embedded within programming language text. SQL statements and associated variable declarations in the host language text are enclosed within "**EXEC SQL**" and ";." These embedded SQL statements and associated declarations are preprocessed into suitable programming language syntax often consisting of procedure calls. The preprocessed source text then becomes input to the programming language compiler or interpreter.

#### Dynamic SQL

Often, a SQL statement required by a program is not known when a program is compiled or even at the beginning of its execution. The need for a particular SQL statement may come about during program execution. SQL'92 includes the concept of *dynamic* SQL. A certain

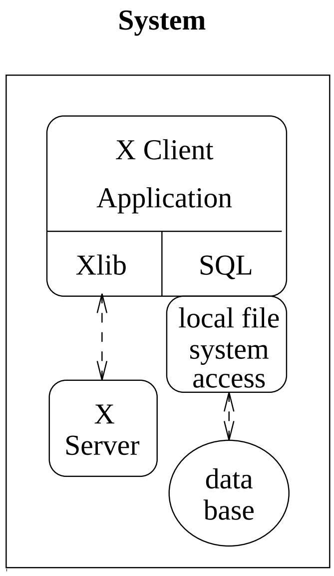



Figure 8.1: SQL on a standalone system.

class of SQL statements may be executed at run time without having been passed through a preprocessor before program compilation like embedded SQL and without having been linked with procedures defined by the module language after compilation. This is accomplished by the Dynamic SQL capability which, given a SQL statement that can be executed at program run time, executes that statement immediately or prepares the statement for execution at some later time.

### 8.1.2 SQL on a Standalone System

SQL provides a standard language which can be used by an application to define and access databases. Figure 8.1 illustrates an application which uses SQL. This application is shown using X as its user interface. The X client application provides the SQL implementation with SQL statements and the SQL implementation provides the responses. The X application interacts with the SQL implementation by using either Embedded SQL or an application programming interface defined by the SQL Module Language.

The SQL implementation interacts with the operating system which provides the means of accessing local files containing the database. The SQL implementation may accomplish local file system access in several ways. It may make use of the normal file system access procedures provided by the host operating system and used by end user applications, or it may directly access the mass storage device forsaking normal file system access procedures, or it may use a combination of the two techniques. By not using the normal file system access procedures, some SQL implementations attempt to improve performance.

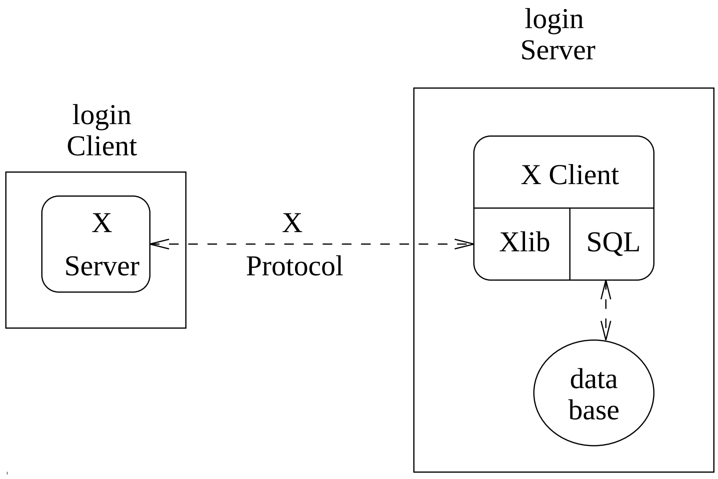



Figure 8.2: SQL with remote login.

### 8.1.3 Basic Security Model

The basic security model of SQL consists of three entities: objects, actions, and users. Objects are defined in the database schema. In SQL'89, the objects are tables, views, columns of tables, and columns of views. In SQL'92, the objects also include domains and assertions. In SQL3, objects will include user defined constructs.

Actions are the operations performed on objects. Actions include: select, insert, delete, update, and references. Users invoke actions on objects.

A privilege is an authorization to a user of an action on an object. A privilege is a 5-tuple:

(grantor, grantee, object, action, grantable)

The *grantor* is a user who is authorized to perform the *action* on *object* and who is authorized to grant the authorization to perform the *action* on *object* to other users. The *grantee* is the user who receives the authorization to perform *action* on *object* from the *grantor*. The true/false flag *grantable* indicates whether the *grantee* is authorized to pass the authorization for performing *action* on *object* to other users.

At object creation time, a user is designated as the *owner* of the object. An owner is authorized to perform all actions on the object and to grant privilege to other users. No user, other than the owner, may perform any action on the object unless that privilege is granted by the owner or by a another user to whom the owner granted the privilege. The owner of the object, or another user granted that privilege by the owner, may revoke the privilege at any time. At that time, the privilege is revoked for the grantee and for any user which obtained the privilege from the grantee.

### 8.1.4 SQL in a Network Environment

In a network environment, SQL may be used in three ways:

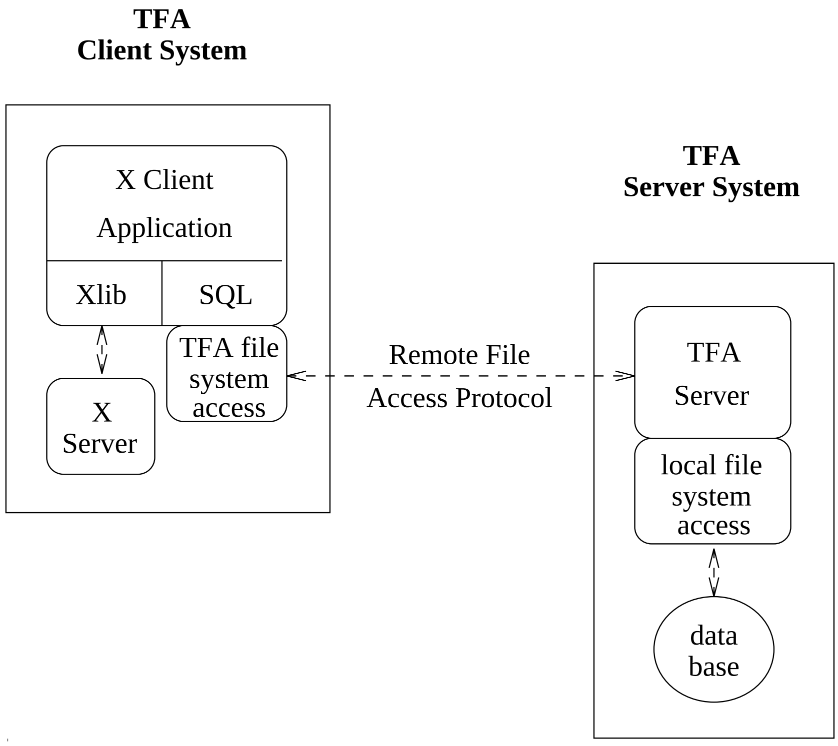



Figure 8.3: SQL with transparent file access.

- logging into a remote system and accessing SQL on the remote system
- mounting the database files located on a remote system and accessing those files by means of SQL on the local system
- using the RDA protocol with SQL specialization to access a remote SQL server

The first two approaches apply generic network access methods to SQL use. The third approach consists of using a network protocol specifically applicable to SQL.

### SQL with Remote Login

In figure 8.2, a user on a client workstation logs into a remote system on which resides the SQL implementation and the database files. From the login client, the user runs an application on the login server which generates SQL statements. These SQL statements provide the access to the database on the login server. A *tty* style login may be used or, as illustrated in figure 8.2, the SQL application may be a X client application running on the login server. This X application drives the login client's keyboard and screen by means of the X server which is executing on the login client.

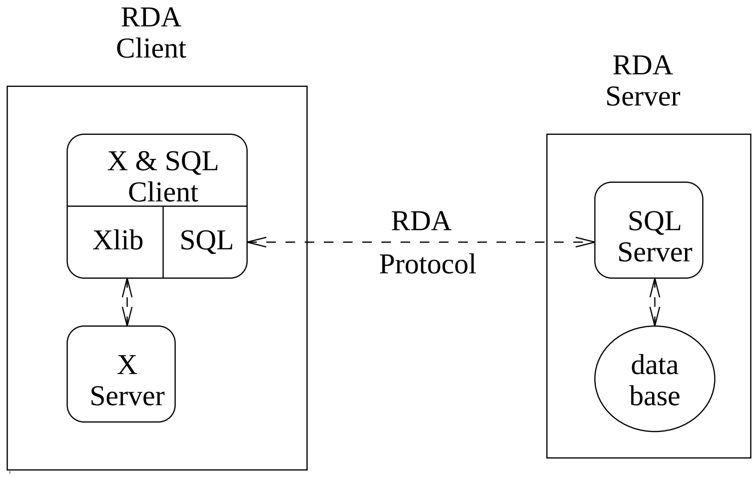



Figure 8.4: SQL with the RDA protocol.

### SQL with Transparent File Access

Transparent file access (TFA) refers to the capability that provides an application with access to remote files as though they were local. The network file system NFS is an example of a protocol which provides a transparent file access capability. With such a mechanism, an application can usually be applied unchanged to files within a file system mounted from a remote file server. This is possible only if the application's access to files is by means of normal file system access procedures provided by the operating system for use by end user applications. A transparent file access capability is generally not available to applications which bypass normal file system access procedures and perform operations on files by directly accessing the mass storage device.

An application that uses a SQL implementation which accesses files exclusively by means of normal file system access procedures is able to access remote files by means of a transparent file access protocol. Figure 8.3 shows how this is accomplished. From the local workstation, a user mounts a remote file system located on a remote server. The database files needed by the SQL implementation on the TFA client are located within this remotely mounted file system. The X client application using the SQL implementation receives responses to its SQL statements as though the database files were local.

Some implementations of SQL do perform operations on files by directly accessing the mass storage device. Consequently, an application which uses such a SQL implementation is unable to make use of a transparent file access mechanism.

### SQL with the RDA protocol

A user may run an application on a client workstation which sends SQL statements to a remote system by means of the RDA protocol. The remote system (called a SQL server) returns the results of the SQL statements to the client workstation. As illustrated in figure 8.4, the application on the RDA client is a X and RDA client application which drives the RDA client's keyboard and screen by means of the X server executing on the RDA client workstation. The protocol, which specifies how SQL statements are communicated to the SQL server and how the SQL server returns results, shown in figure 8.4 is RDA.

RDA is specified in ISO 9579. RDA is an application layer protocol in an OSI stack. The RDA standard is in two parts: Generic RDA and SQL specialization. Generic RDA specifies a protocol which can support database operations but does not specify the syntax or semantics of the database operations sent from client to server. The SQL Specialization defines how SQL is to be used in conjunction with Generic RDA.

RDA fits within an OSI stack in one of two ways. In the Basic Application Context, RDA is supported by ACSE and is based on a simple client/server model. This model only describes how a single SQL client interacts with a single SQL server. The Basic Application Context does not support distributed database applications. In order to support distributed database applications, the Transaction Processing (TP) Application Context must be used. In the TP Application Context, RDA is supported by ISO TP (ISO 10026-3) and ISO CCR (ISO 9805).

### 8.1.5 Security with SQL in a Network Environment

The security mechanism available with SQL in a network environment includes the same basic security model described in section 8.1.3. This mechanism depends upon correct user identification in order to be effective. When SQL is used on a network by remote login, user authentication is provided by the remote login mechanism. When SQL is used on a network by means of transparent file access, user authentication is provided by the login mechanism on the TFA client and file access is controlled by the TFA server. When SQL is used on a network by means of RDA, it is the SQL'92 statement *connect* which contains a character string which identifies the user. Currently, this character string is the only means available within the SQL and RDA specifications to attach user identification and credentials.

Several implementations of SQL in a network environment are currently marketed. However, these implementations do not use the RDA protocol. Each uses its own protocol so that a SQL client from one vendor and an SQL server from another vendor do not interoperate. These implementations may use several different protocols to support their SQL network implementation. These protocols include TCP/IP, DECnet, and SNA.

Within some of these implementations, the user name and password make up the user identification string in the SQL *connect* command and this string is passed in plain text across the network. From a security point of view, that this string is passed in plain text is not good practice.

SQL, in the specification of its use with RDA and in most of its implementations in a network environment, is dependent on external support for security mechanisms. See chapters 9 and 10 for a description of vulnerabilities and security mechanisms in a network environment.

# Part V

# Network Services Security

# Chapter 9

# Network Security Threats

**Karen Olsen**

## 9.1 Generic Description of Threats

A threat is a circumstance, condition, or event with the potential to cause harm to personnel and/or network resources in the form of destruction, disclosure, modification of data, denial of service, and/or fraud, waste, and abuse. This chapter describes the most common security threats to network systems. Network security threats include impersonation, eavesdropping, denial of service, packet replay, and packet modification. For a more detailed presentation addressed to a technical specialist with in-depth knowledge of network protocols, see [CB94].

### 9.1.1 Impersonating a User or System

As described in section 6.1, common ways to identify and authenticate users include the use of physical keys, account names and passwords, and biometric checks. Password guessing, password trapping, use of security holes in programs, and use of common network access procedures are methods that can be used to impersonate users. Impersonation attacks involving the use of physical keys and biometric checks are less likely.

Compared to standalone systems, systems on networks are much more susceptible to attacks where crackers impersonate legitimate users for the following reasons:

- Crackers have potential access to a wide range of systems over a large geographic area. As a result, network nodes that are not securely configured and/or are running programs with security holes are particularly vulnerable.

- A cracker can use the *finger* or *ruser* programs to discover account names and then try to guess simple passwords (see sec. 9.2.7).

- Crackers can make use of more sophisticated password guessing methods, e.g., a cracker could use a distributed password guessing program in which multiple systems are used to guess passwords.
- Electronic eavesdropping can be used to trap user names and unencrypted passwords sent over the network (see sec. 9.1.2).
- Common network access procedures (see sec. 9.2) can be used to impersonate users. Attacks where root privileges are gained are particularly dangerous because a cracker may be able use common network access procedures to break into numerous systems.
- Crackers can monitor the activity on a system and impersonate a user when the impersonation attacks is less likely to be detected.

Individual systems on a network are also vulnerable to imposter attacks. A cracker can configure a system to masquerade as another system, thus gaining unauthorized access to resources or information on systems that "trust" the system being mimicked. section 10.2.7 discusses how to protect a system against impersonation when using the "r" commands.

### 9.1.2 Eavesdropping

Eavesdropping allows a cracker to make a complete transcript of network activity. As a result, a cracker can obtain sensitive information, such as, passwords, data, and procedures for performing functions. It is possible for a cracker to eavesdrop using wiretapping, eavesdropping by radio and eavesdropping via auxiliary ports on terminals [GS91]. It is also possible to eavesdrop using software that monitors packets sent over the network. In most cases, it is difficult to detect that a cracker is eavesdropping.

Many network programs, such as *telnet* and *ftp* are vulnerable to eavesdroppers obtaining passwords which are often sent across the network unencrypted. Threats associated with use of *telnet* and *ftp* are described in sections 9.2.1 and 9.2.2.

Network programs which involve file transfer are susceptible to eavesdroppers obtaining the contents of files. In particular, NFS, RPC, *rcp*, and *ftp* are vulnerable to unintended disclosure of data. Encryption can be used to prevent eavesdroppers from obtaining data traveling over unsecured networks. Section 5.1 provides information on cryptography.

### 9.1.3 Denial of Service

Multi-user, multi-tasking operating systems are subject to "denial of service" attacks where one user can render the system unusable for legitimate users by "hogging" a resource or damaging or destroying resources so that they cannot be used. Denial of service attacks may be caused deliberately or accidentally. Taking precautions to prevent a system against unintentional denial of service attacks will help to prevent intentional denial of service attacks.

Systems on a network are vulnerable to overload and destructive attacks as well as other types of intentional or unintentional denial of service attacks. Three common forms of network denial of service attacks are service overloading, message flooding, and signal grounding. It is important for system administrators to protect against denial of service threats without denying access to legitimate users. In general, denial of service attacks are hard to prevent. Many denial of service attacks can be hindered by restricting access to critical accounts, resources, and files, and protecting them from unauthorized users.

### 9.1.4 Packet Replay

Packet replay refers to the recording and re-transmission of message packets in the network. Packet replay is a significant threat for programs that require authentication sequences, because an intruder could replay legitimate authentication sequence messages to gain access to a system. Packet replay is frequently undetectable, but can be prevented by using packet time-stamping and packet sequence counting.

### 9.1.5 Packet Modification

Packet modification is a significant integrity threat which involves one system intercepting and modifying a packet destined for another system. In many cases, packet information may not only be modified, but it may also be destroyed.

## 9.2 Threats Associated With Common Network Access Procedures

A common network access procedure is a method allowing a user to access resources provided by a remote system. There are several threats, ranging from eavesdropping to destruction of data, that are identified with common network access services. Common network access services and their threats are discussed in the following sections. It is not uncommon for a cracker to install an altered version of a common network access program. The altered program may accept a special input sequence for the purpose of spawning a shell for the cracker, or, the altered program may be used to capture passwords.

### 9.2.1 Telnet

The TELNET protocol allows a user to log into a system over the network and use that system as though the user was sitting at a terminal that was directly connected. The client and server programs which use the TELNET protocol are *telnet* and *telnetd*, respectively. The *telnet* command provides a user interface to a remote system. If *telnet* is invoked with the name of a remote host as an argument, a prompt is displayed and a user can log in as if they had called the system with a modem. Logging into a system using *telnet* can pose a security risk because a username and password are sent over the network in plain text one character per packet. Since these characters are not encrypted, it is possible for an electronic eavesdropper to capture a username and password for a system for which a *telnet* connection is being established.

In addition to the danger of network snooping, using the TELNET protocol presents the same sort of security risks as dial-in modems. *"Practical UNIX Security"* [GS91] lists the following reasons why using the TELNET protocol with a wide area network poses more risks than those posed by modems.

- Few computer centers publish the telephone numbers of their system's modems. For systems on the Internet and listed in the Internet domain servers, a user only needs to know a system's name in order to connect via *telnet*. Although this makes access easier for authorized users, it also makes access easier for attackers.

- Because it is significantly faster to connect via *telnet* to a system than to call the system with a modem, an attacker can try to guess more passwords in any given amount of time.

- Long distance calls cost the caller money, but there is usually no incremental charge for using *telnet* over the Internet. As a result, systems on the network are more subject to attack from around the country and around the world.

- It is often easier to access a system anonymously on the Internet than over phone lines. Modern telephone switching systems can trace calls in seconds and in many cases deliver the calling number to the system. Internet protocols make it easier for an intruder to disguise the source of an attack.

### 9.2.2 File Transfer Protocol

The File Transfer Protocol (FTP) allows users to connect to remote systems and transfer files back and forth. FTP is implemented by the *ftp* client program and the *ftpd* server program. As part of establishing a connection to a remote machine, *ftp* relies on a username and password combination for authentication. Use of *ftp* poses a security problem similar to use of the TELNET protocol because passwords typed to *ftp* are transmitted over the network in plain text, one character per packet. These packets can be intercepted. Use of versions of *ftpd* older than the most recent version pose security threats because older versions have bugs that allow crackers to break into a system.

Another problem area for *ftp* is "anonymous *ftp*." Anonymous *ftp* allows users who do not have an account on a machine to transfer files to and from a specific directory. This capability is particularly useful for software or document distribution to the public. To use anonymous *ftp*, a user passes a remote computer name as an argument to *ftp* and then specifies anonymous as their username.

One of the problems with anonymous *ftp* is that there is often no record of who has requested what information. Another problem with anonymous *ftp* is the threat of denial of service attacks. For deliberate or accidental denial of service attacks, authorized users may be denied access to a system if too many file transfers are initiated simultaneously. It is important to securely set up the anonymous FTP account on the server because everyone on the network will have potential access. If the anonymous ftp account is not securely configured and administered crackers may be capable of adding and modifying files. section 10.2.1 describes techniques which should be used to increase security when using *ftp*.

### 9.2.3 Trivial File Transfer Protocol

The trivial file transfer protocol (TFTP) is a UDP-based file transfer program that is frequently used to allow diskless hosts to boot over the network. TFTP is implemented by the *tftp* client program and the *tftpd* server program. Because TFTP has no user authentication, it may be possible for unwanted file transfer to occur. It is a significant threat that *tftp* may be used to steal password files. section 10.2.3 describes a way to verify that a system is not using a version of *tftpd* with know security holes. In particular, versions of SunOS prior to release 4.0 are known to have a security hole because the *tftpd* program did not restrict file transfer.

### 9.2.4 Mail

Electronic mail is a valuable service which allows users to send and receive messages across networks. On most versions of Berkeley-derived Unix systems, the *sendmail* program is used to enable the receipt and delivery of mail. Older versions of *sendmail* have several bugs that allow security violations.

A few precautions can be taken to ensure secure operation of *sendmail*. These precautions are discussed in section 10.2.4 The UNIX *mail* command understands UUCP-style addressing. For information on using UUCP refer to sections 9.2.5 and 10.2.5.

### 9.2.5 Unix-to-Unix Copy System

The *U*nix-to-*U*nix Co*P*y system (UUCP) is a collection of programs primarily used for transferring files between UNIX systems, sending mail to users on remote systems, and executing commands on remote systems. For a more detailed description of aspects of UUCP related to computer security consult [GS91]. For a complete description of UUCP consult a UNIX vendor's manuals, e.g., [SUN90b].

It is possible for UUCP to be used for unauthorized access. Section 10.2.5 describes a few ways to make UUCP more secure.

### 9.2.6 rlogin, rsh, and rcp

The *rlogin*, *rsh*, and *rcp* programs are often referred to as the "r" commands, where "r" stands for remote. The *rlogin* program establishes a remote login session to a remote system, the *rsh* program connects to a remote system and executes a specified command, and the *rcp* program copies files between systems. Section 10.2.7 describes how to minimize impersonation attacks and how to provide as much security as possible when using the trusted hosts facility associated with the "r" commands.

The concept of "trusted" hosts makes use of these commands convenient. Each remote machine may have a file named */etc/hosts.equiv* containing a list of trusted hostnames. Users with the same username on both the local and remote system may remotely login from the systems listed in the remote system's */etc/hosts.equiv* file without supplying a password.

Trusted hosts pose a security threat because the host authentication mechanism can be defeated. In addition, the users on that host cannot alway be trusted. If a cracker manages to break into an account on a host, and that host is trusted by other systems, the user's accounts on all the other systems are compromised.

Individual users may set up a similar private equivalence list with the file *.rhosts* in their home directories. Each line in this file contains a hostname and a username separated by a space. An entry in a user's remote *.rhosts* file permits the user who is logged into the system specified by hostname to login to the remote system without supplying a password. Use of *.rhosts* files is a security threat because an administrator is unable to exclusively control access to the system via the "r" commands. Users are more likely to tailor their *.rhosts* files more for convenience than for security.

Trusted hosts involve a security risk for accounts which have an asterisk in the encrypted password field of the password file. Since the trusted hosts facility bypasses password checking, accounts that have login disabled could be accessed using "r" commands if either the *hosts.equiv* file or *.rhosts* file grant permission.

When using *rlogin*, if the name of the local host is not found in the */etc/hosts.equiv* file on the remote system, and the local username and hostname are not found in the remote user's *.rhosts* file, then the remote system will prompt for a password. Prompting for a password is a threat because the password is sent unencrypted in a single packet over the network.

Trusted users on trusted hosts are allowed to execute *rsh* and *rcp* commands without requiring a password to be entered. The *rsh* program will not prompt for a password if access is denied on the remote system unless the command argument is omitted. If a command argument is not specified for *rsh*, *rsh* logs the user onto the remote system using *rlogin*. For users on hosts that are not trusted (i.e., neither listed in the */etc/hosts.equiv* file nor the *.rhosts* file), *rcp* does not prompt for passwords.

Although the trusted hosts concept provides convenience for authorized users, systems not properly administered are vulnerable to unauthorized access. Systems whose */etc/hosts.equiv* or */etc/hosts.lpd* files contain a "+" are extremely vulnerable because the "+" entry means that the system trusts all other systems. Similarly, if any */.rhosts* file contain a "+ +" entry, the system is vulnerable to access by non-trusted users on non-trusted systems. On Sun systems, the single entry "+" is contained in the default *hosts.equiv* file in the distribution, thus trusting all hosts. This is clearly a security problem.

### 9.2.7 Commands Revealing User Information

Commands which reveal user and system information pose a threat because crackers can use that information to break into a system. This section provides a brief description of various commands whose output makes a system vulnerable to break-ins.

#### finger

The "finger" service provided by the *finger* client program and the *fingerd* server program displays information about users. When *finger* is invoked with a name argument, the */etc/passwd* file is searched and for every user with a first name, last name, or username that matches the name argument, information is displayed. When the *finger* program is run with no arguments, information for every user currently logged onto the system is displayed. User information can be displayed for remote machines as well as for the local machine.

The output of *finger* typically includes login name, full name, home directory, last login time, and in some cases when the user received mail and/or read mail. Personal information, such as telephone numbers, are often stored in the password file so that this information is available to other users. Making personal information about users available poses a security threat because a password cracker can make use of this information. In addition, *fingerd* can reveal login activity.

Versions of *fingerd* older than November 1988 are vulnerable to abuse because they contain a bug.

#### rexec

The *rexec* and *rexecd* programs allow remote execution. Unlike *rlogin*, *rsh*, and *rcp*, *rexecd* does not use the trusted host mechanism. A client transmits a message specifying the username, the password, and the name of a command to execute. The *rexecd* program is susceptible to abuse, because it can be used to probe a system for the names of valid accounts. In addition, passwords are transmitted unencrypted over the network.

#### rwho, rusers, netstat, and systat

Information obtained from *rwho*, *systat*, and *netstat* may be valuable to potential crackers. The *rwho* and *rusers* commands reveal who is logged in on remote machines. Idle times can also be displayed. The *netstat* command displays the contents of various network-related data structures in various formats. User information can be display by *systat*.

### 9.2.8 Distributed File Systems

The Network File System (NFS) and the Remote File Sharing (RFS) service are distributed file systems which use a client/server model to allow computers to share files over the network. NFS and RFS allow users to access, read, and change the contents of files stored on servers without having to log into the server or supply a password. This feature results in several security problems.

In order to share files using a distributed file system, directories of files must first be exported by the server. Once the client has mounted the exported files, users may access these files as though the files were stored locally. There are many threats associated with using distributed file systems, especially if no precautions are taken when exporting directories from the server.

### Network File System (NFS) Threats

The Network File System (NFS) is a stateless protocol which uses remote procedure calls (RPC) built on top of the external data representation (XDR) protocol [SUN90a]. NFS provides most of the properties of a UNIX file system and can be implemented on almost any operating system. Threats associated with using NFS include the following:

- If a directory is exported with no access list specified, any system on the network is capable of accessing the exported files.

- If a directory is exported with root access given to specified clients, anyone with superuser privileges on one of the clients can modify files on the server owned by root.

- An NFS server grants file access to users on clients that have user ID and group ID mappings which correspond to the server, i.e., a user on a client who has a user ID of 100 can access files on the server that are owned by user ID 100 and have the proper read, write, or execute permission bits for owner set. This is a threat because it is easy for one user to impersonate another, especially if the user has superuser privileges on the client.

- It is possible for a client to be impersonated, especially if the client is a system that is turned off regularly.

- NFS uses file handles to reference files. It is relatively easy to guess valid file handles because file handles consist of a file system id and and inode number. It is possible to increase the difficulty of guessing a valid file handle by using a program to randomize the inode of each file.

As is often the case, a vendor may distribute NFS with no security features enabled. Section 10.2.8 describes ways to export files so that threats of unwanted file access and manipulation are reduced. Other techniques for improving the security when using NFS are also discussed.

### File Permissions

File permissions indicate what kind of access is granted to users on a system. There are three types of permission (the ability to read, write, or execute) and there are three categories of users (the file's owner, users who are in the file's group, and everybody else on the system, with the exception of the superuser). When using NFS, and when using the underlying RPC protocol, it is possible for unauthorized users to obtain unintended access to files.

It should be noted that the semantics of using commands which change the permissions mode of a file, the owner of a file, or the group ownership of a file differ slightly when using NFS than when using a local file system. Often with distributed file systems, caching is used to increase performance. If caching is used, then the effects of changing the permissions mode of a file, the owner of a file, or the group ownership of a file may not take effect immediately on the remote file system.

### Remote File Sharing (RFS)

Remote File Sharing (RFS) is a distributed file system provided with most System V-based systems [ATT90]. RFS is also supported by more recent versions of SunOS. Unlike NFS which provides a generic file system, RFS provides an exact copy of a UNIX file system. Another difference between NFS and RFS is that RFS groups hosts into *domains* for facilitating mounting of file systems. For the most part, security threats associated with NFS are also associated with RFS.

This section lists threats associated with various aspects of RFS. Section 10.2.9 lists ways to make RFS more secure. RFS provides four levels of security to protect resources [Cur92].

- Connect Security

  Before attempting to mount remote resources the local system must first set up a connection to the server. For many systems, *rfadmin* is the RFS verification command used to restrict access to a given set of machines. This command specifies a password which must be entered before a system is allowed to connect to a server. If a password has not been provided for a system, that system is allowed to connect to the server without a password check. This poses a threat of unintended access, especially if precautions were not taken when exporting files.

- Mount Security

  Once a connection has been established between a system and a server the system may mount any file systems that the server has exported. For System V Release 4 version of RFS, *share* is used to export file systems. Appropriate options should be specified for *share* so that unintended access is not granted for resources.

- User and Group Mapping

  As a method of controlling access to resources, a system administrator is able to create user and group id mappings by editing the files *uid.rules* and *gid.rules*. These files allow global rules and host-specific rules to be specified. The threat of denial of service may result if user and group mappings are set up in such a way that users are not able to access their own files. On the other hand, poorly defined user and group mappings may allow unintended access to resources.

- Regular UNIX File Permissions

  UNIX file permissions that are improperly set can allow unintended access for local users of a system. Unintended access can also be threat when files are exported with improperly set file permissions.

### 9.2.9 Network Information Service

Network Information Service (NIS) [SUN90b], formerly called Yellow Pages (YP), is a distributed database system that lets systems share password files, group files, host tables, and other files over the network. NIS simplifies the management of a network because all of the account and configuration information is reconstructed and stored on a single computer, the NIS master server. NIS is included with SunOS, most SVR4 UNIX systems, and many other flavors of UNIX.

Shared NIS database files are called maps and hosts that belong to the same NIS domain share the same set of maps. NIS slave servers, which obtain up-to-date copies of the maps from the NIS master server, are used to provide information when the NIS master server is down. Although NIS simplifies the task of system administration, it also presents several security problems when it is not securely configured.

NIS naming services were originally designed to address the administration requirements of client/server networks in the 1980s. Such networks had specific characteristics, including [JS92]:

- Their size seldom exceeded a few hundred multivendor client desktops and a few general-purpose servers.
- They spanned at most a few geographically remote sites.
- They had friendly, trusted, and sophisticated users and security was not an issue.

Since NIS was not designed to address security requirements, NIS is susceptible to abuse. The following is a list of threats associated with using NIS [Cur92]. Section 10.2.10 discusses methods which can be taken to avoid potential security problems with NIS.

- The file *hosts.equiv* is one of the many files that can be controlled with NIS. Systems that come with NIS software from Sun Microsystems are distributed with the default *hosts.equiv* file containing a "+" as its single entry. This is a threat because the default *hosts.equiv* file considers all hosts to be trusted.
- NIS works by having either of the lines "+::0:0:::" or "+:" in the password or group file. When a program reads the password or group file and encounters a line with a "+" as the first character, the plus sign indicates that the program needs to ask the NIS server for the remainder of the file. Using the "+::0:0:::" format is a threat because for some systems, if the leading "+" is carelessly deleted, an attacker can log in with a null login name and gain superuser access to the system.

- The *ypset* command can be used to tell a process called *ypbind* that NIS requests should be sent to a specific host. This feature was designed to allow debugging and to allow hosts that are not on a network with an NIS server to use NIS. The *ypset* command presents a security problem because it can be used to direct requests to a fake NIS server.

- Certain versions of NIS map-building procedures leave the maps world-writeable. World-writeable maps pose a threat because anyone is capable of changing the contents of the maps to invalid information.

- Any user is capable of obtaining copies of the databases exported by a NIS server. This can result in unintended disclosure of the distributed password file and all the other information contained in the NIS database.

Network Information Services Plus (NIS+), incorporated into Solaris 2.0 (SunOS 5.0), replaces NIS. NIS+ enhancements include support for hierarchical domain names, use of a new database model, and changes to the NIS authentication and authorization model [JS92]. NIS+ contains security aspects lacking in NIS.

# Chapter 10

# Improving Security in a Network Environment

**John Barkley**
**Lisa Carnahan**
**Karen Olsen**
**John Wack**

Most operating systems have little or no security enabled when initially installed. In order to provide system security, a system administrator must not only be knowledgeable of the different ways to protect a system, but it is also imperative that the administrator implement the computer security plan in a thorough and consistent manner. Section 10.1 lists many of the aspects of security that need to be considered when protecting a system, regardless of whether the system is standalone or connected to a network. Networked systems are much more vulnerable to break-ins because of their accessibility over the network and the use of inherently insecure network protocols. In addition, if one system on a network is broken into, then other systems on the network may be compromised. See [CB94] for a detailed presentation of network security threats and suggestions for improving network security addressed to the technical specialist with in-depth knowledge of network protocols.

Many network protocols are subject to abuse. Section 10.2 describes precautions that can be taken to improve security for common network access procedures. Each system using common network access procedures must take precautions to protect against the threats associated with each service used. For example, each system running the Unix "r" commands must take precautions to prevent the threats associated with the trusted hosts facility (see sec. 9.2.6) from being exploited. Individual systems must be responsibly administered so that all systems cooperate to achieve a secure network.

Secure gateways (see sec. 10.3) provide network security by blocking certain protocols and services from entering or exiting subnets. Secure gateways, or *firewalls*, have an advantage over the methods described in section 10.2 because security can be concentrated on a firewall. The firewall can be used to filter commonly exploited common network access protocols from entering a subnet while permitting those common network access protocols to be used on the inside subnet without fear of exploitation from outside systems.

Robust authentication mechanisms improve the authentication process beyond conventional authentication mechanisms such as passwords. Section 10.4 discusses robust authentication procedures.

## 10.1 Administering Standalone Versus Networked Systems

Security precautions that should be taken when administering standalone systems also apply to networked systems. Although a discussion of security threats for standalone systems is out of the scope of this report, the following is a list of several security precautions to consider when administering a system regardless of whether the system is standalone or connected to a network.

- Avoid weak passwords, i.e., passwords that are easy to crack.
- Make use of file access control, auditing, and backups.
- Check with vendors and install all applicable security-related patches.
- Limit readability and writeability of system files.
- Regularly check system binaries against copies from distribution media to verify that programs have not been modified. Binaries for common network access procedures, such as *rlogin*, *rsh*, *rcp*, *ftp*, *telnet* and *uucp* are particularly vulnerable. Altered versions of these binaries can allow unauthorized access to the system.
- Examine all commands or scripts that run automatically at specified dates and times, e.g., for SunOS *cron* and *at* can be used to execute commands and scripts at specified dates and times. These commands could be useful to a cracker.
- Check for unauthorized setuid and setgid programs, i.e., check for programs that grant special privileges to the user who is executing the program.
- Protect modems and terminal servers.

It is common for workstations to be primarily used by an individual user. As a result, individual users are forced to become system administrators. Users of individual systems may either not have the knowledge to securely configure their workstation, or may decide to sacrifice security for convenience. In order to protect against unauthorized use, systems should be responsibly administered, regardless of whether they are standalone, or networked single-user or multi-user systems.

## 10.2 Improving Security of Common Network Access Procedures

Most programs which provide network services are susceptible to abuse. The Network File System (NFS) and commands which use the trusted hosts facility are particularly vulnerable. A system administrator should be aware of security risks associated with the use of each service. For systems that do not need to provide a specific network service, the system administrator may want to consider disabling the appropriate program. For example, a system administrator may want to disable the *ftp* and *ftpd* programs for a system that has no need for file transfer service.

This section describes ways to prevent vulnerabilities of common network access procedures from being exploited. For stronger security, the methods described in this section can be combined with the use of secure gateways (see sec. 10.3) or robust authentication methods (see sec. 10.4).

In addition to describing secure gateways, section 10.3 also describes a third-party package called the "TCP Wrapper" package. This package serves as a front end which provides access control for all services executed from the UNIX inet daemon process. The TCP Wrapper package can be used to determine whether a host requesting a telnet, ftp, or "r" command connection is authorized, to log the request, and then either to accept or reject the connection.

In order to provide an overall secure network, all systems using common network access procedures must be protected against the threats described in section 9.2. If individual systems do not protect themselves from exploitation of inherently flawed common network access procedures and a system is broken into, then other systems on the network may be compromised.

### 10.2.1 The "r" Commands Versus telnet/ftp

There are a few security related tradeoffs between using *telnet* and *ftp* versus using the "r" commands. For example, *rlogin*, *rsh*, and *rcp* are less susceptible to the eavesdropping of passwords than *telnet* and *ftp* because with the "r" commands, a user does not need to type in a password. An exception is when *rlogin* is invoked for a non-trusted user. In this case, a single packet containing the user's password will be passed over the network. However, in general, *telnet* and *ftp* are more susceptible to interception of user names and passwords than *rlogin*. Note that, except for the case when *rsh* invokes *rlogin*, *rsh* and *rcp* never prompt for passwords.

The use of trusted hosts when using the "r" commands introduces security problems which are not relevant when using *telnet* and *ftp*. Trusted hosts introduce security problems because the host authentication mechanism can be defeated, and users on a trusted host cannot always be trusted. If an attacker manages to break into an account on a host, and that host is trusted by another computer, the user's account on the other computer is compromised. In addition, the *.rhosts* file can be compromised by an attacker by adding entries that permit access by others.

Thus, the basic tradeoff between the use the "r" commands versus *telnet/ftp* is whether it is more insecure to permit trusted hosts configured so that passwords do not go across the network in plain text versus having passwords passing across the network in plain text. Neither situation is desirable in general. It is up the system administrators and the network administrators to choose the *better* approach for their environments. For example, in general, it is harder to eavesdrop on a token ring network than on an ethernet network. So, the choice on a token ring network may be to use *telnet/ftp*.

### 10.2.2 Improving the Security of FTP

Use of the *ftp* and *ftpd* programs pose several security problems unless precautions are taken when configuring and administering this service. This section describes several techniques for improving security when using *ftp* and *ftpd*.

As mentioned in section 9.2.2, older versions of *ftpd* had several bugs that allowed crackers to break into a system. To minimize the threat of a break-in, the most recent version of *ftpd* should be used.

It is desirable to restrict certain remote users from accessing files. The */etc/ftpusers* file contains a list of users who are not allowed to use FTP to access any files. At a minimum, the */etc/ftpusers* file should contain all accounts, such as root, uucp, news, bin, ingres, nobody, daemon that do not belong to human users.

Setting up anonymous FTP may vary for different implementations. Below is a description of guidelines that can be followed to minimize unintended use of anonymous FTP [CA-93].

1. Disable the ftp account by placing an asterisk in the password field of the password file. This will prevent users from logging onto the system using a user name of ftp.

2. Verify that the anonymous FTP root directory and its subdirectories are not owned by the ftp account and are not in the same group as the ftp account. Anonymous FTP subdirectories generally include *ftp/bin*, *ftp/etc*, and *ftp/pub*. These directories should be write protected.

3. In order to print user names and group names when files are listed, a *passwd* and *group* file are needed in the *ftp/etc* directory. To prevent crackers from obtaining copies of the system's */etc/passwd* and */etc/group* files, a dummy copy of each file should be used. All password fields should be changed to asterisks.

4. An administrator should not provide writeable anonymous ftp directories unless the threats of providing writeable directories are known and precautions are taken to minimize these threats.

Several other precautions should be taken. It is possible for remote users to transfer large files to the ftp/pub directory. This can cause a disk partition to become full. To prevent this problem, put a file quota on the user ftp, or locate the ftp account's home directory on an isolated partition. The contents of the pub directories should be monitored and any suspicious files should be deleted.

Previously, hosts have been temporarily rendered unusable by massive numbers of FTP requests. If these incidents were deliberate, they would be considered a successful denial of service attack. Load-limiting techniques can help to avoid such problems.

### 10.2.3 Improving the Security of TFTP

As mentioned in section 9.2.3, TFTP is a UDP-based file transfer program that provides no security. The TFTP program is allowed to transfer a set of files to any system on the Internet that asks for them. TFTP is often used to allow diskless hosts to boot from the network. Because TFTP lacks security, *tftp* is usually limited to transferring files only to or from a certain directory. Early versions of *tftp* did not impose file transfer restrictions. In particular, versions of SunOS prior to release 4.0 did not restrict file transfer from *tftp*.

The following procedure can be used to test a system's version of *tftp* for security problems [GS91]:

**tftp localhost**
**tftp> get /etc/passwd tmp**
**Error code 1: File not found**
**tftp> quit**
%

If *tftp* either hangs with no message or does not respond with "File not found" and instead transfers the file, *tftp* should be replaced with a current version.

### 10.2.4 Improving the Security of Mail Services

The following precautions should be taken to ensure secure operation of *sendmail* [GS91]:

1. Verify that the version of *sendmail* used is recent. Older versions of *sendmail* have several bugs that allow security violations.

2. Remove the "uudecode" and "decode" alias from the aliases file. This file is usually */etc/aliases or /usr/lib/aliases.*

3. For aliases that allow messages to be sent to programs, make sure that there is no way to obtain a shell or send commands to a shell from these programs.

4. Verify that the "wizard" password is disable in the configuration file *sendmail.cf*.

5. Verify that *sendmail* does not support the "debug" command. This can be done with the following commands:

% **telnet localhost 25**
Connected to localhost
Escape character is ``^]''.
220 hostname sendmail 5.61 ready at Fri, 18 Sep 92 15:10:48 EDT
**debug**
500 Command unrecognized
quit
%

If *sendmail* responds to the "debug" command with the message "200 Debug set", then *sendmail* is vulnerable to attack and should be replaced with a newer version.

### 10.2.5 Improving the Security of UUCP

If UUCP is not properly installed, a system's security can be compromised. The following is a list of ways to configure UUCP more securely [GS91]:

- If there is not a need for UUCP services, delete or protect the UUCP system.
- The *uucico* program, a file transport program for the UUCP system, must log into a system in order to transfer files or run commands. Assigning a password to the *uucp* account can deter crackers from logging in.
- Create additional */etc/password* entries for each system that calls your system. Having different logins for each remote system allows an administrator to grant different privileges and access to different remote systems.
- If desired, required callback for certain systems, to deter impersonation attacks.
- Configure UUCP so remote systems can retrieve files only from specific directories.
- If file retrieval is not needed, disable remote file retrieval.
- UUCP control files should be protected so that they cannot be read or modified using the UUCP program.
- Limit the commands which can be executed off site to those that are absolutely necessary.
- To protect information in the *L.sys* (Version 2) or *Systems* (Basic Networking Utilities version) log files from being misused, the appropriate file should be owned by the *uucp* user and be unreadable to anybody but UUCP.

The following is a description of three main UUCP security problems:

1. Mail delivery to files can be used to corrupt system databases or application programs. If a system allows mail to be sent to a file, then the mailer is unsecure and the version of UUCP being used should be disabled or upgraded to a current version.

2. The UUCP system should not allow commands to be encapsulated in addresses. If a system executes commands encapsulated in an address then the *uux* program is unsecure and should be upgraded to a current version.

### 10.2.6 Improving the Security of finger

Versions of *fingerd* older than November 1988 contain a bug. Older versions should be replaced with a newer version.

The *finger* command, as well as *rexec*, *rwho*, *rusers*, *netstat* and *systat*, reveal information which may be valuable to potential crackers. Information revealed may be used to monitor login and network activity or to guess passwords. To improve security, it is recommended that theses services be disabled.

### 10.2.7 Improving the Security of the "r" Commands

As mentioned in section 10.2.1, the *rlogin* and *rsh* commands query for a password if either the user or the client system is not trusted by the server to which the *rlogin* or *rsh* is addressed. Under these circumstances, the *rlogin* and *rsh* commands send the password in plain text across the network in a single packet. Consequently, a packet may be intercepted. If a server is not going to make use of the trusted user and trusted host capability of the "r" commands, it is prudent to disable the *rlogind* and *rshd* server programs. A server could support the same functionality of *rlogin* and *rsh* in the absence of trusted users and hosts with *telnet*.

#### Administering Trusted Users and Hosts

Given that a server is going to support the "r" commands using trusted users and hosts, it is important for an administrator to be aware of which hosts and users are allowed to access the system without supplying passwords. The */etc/hosts.equiv* and */etc/hosts.lpd* files should not contain an entry of "+" (plus) unless required in the operating environment and protected by a firewall network configuration [CA-92]. An entry of "+" assumes all hosts to be trusted. Similarly, *.rhosts* files should never contain a "+ +" entry. A *.rhosts* file containing "+ +" will trust all users on all systems.

The "r" command daemons access a *.rhosts* file in a user's home directory on the server as part of the access control process. A user is able to list in this file the trusted hosts of the user's choice. This implies that the administrator is unable to exclusively control access.

The problem of how to administer users' *.rhosts* files has two solutions which are not difficult to implement. The most obvious solution is to have a daemon, which could be a shell script, monitoring the contents of users' *.rhosts* files. Any undesirable trusted hosts in these files could be removed. While this approach can work reasonably well, such monitoring can only take place periodically and leaves open the possibility that undesirable access can exist for short periods of time.

Another approach to controlling use of *.rhosts* files is for the administrator to disable the use of users' *.rhosts* file completely. This is accomplished by the administrator creating a *.rhosts* directory and a file within that *.rhosts* directory where both the *.rhosts* directory and the file in that directory are owned by *root* or the administrator. By excluding the user from all access to the *.rhosts* directory and the file within, the user can neither delete the *.rhosts* directory (note that only *root* may unlink a directory), nor create a *.rhosts* recognized by an "r" command. The administrator can then maintain exclusive control over access by means of the "r" commands through the use of the *hosts.equiv* file.

It is possible for an intruder to modify existing *.rhosts* and */etc/hosts.equiv* files or to create new *.rhosts* files in users' home directories to allow future unauthorized access for the attacker. To prevent against unauthorized modification of files bypassing authentication to trusted hosts and users, an administrator may want to use a daemon which monitors the contents of *.rhosts* files and the contents of */etc/hosts.equiv* as well. An administrator should also verify that any account with login disabled is not accessible by the trusted hosts facility.

Note that the *hosts.equiv* file should not be used to permit access to print service (*lpd*). The *hosts.lpd* file may be used for that purpose. An entry in the *hosts.lpd* file only grants access to print service while an entry in the *hosts.equiv* file grants access to both print service and the "r" commands.

### Protecting Against Impersonation Using the "r" Commands

Once a host becomes trusted, there is the possibility that a trusted host or a trusted user may be impersonated. In the case of trusted user impersonation, there really is not much that can be done. The superuser on a trusted host is capable of impersonating any user. Note that it is usually not prudent to give trusted user access to the superuser. Probably the best way to minimize the risk of trusted user impersonation is to be aware of the quality of the administration on the trusted host.

Impersonating a trusted host is not so easily accomplished. Having two systems with the same IP address active on the same IP subnet usually results in strange behavior on the part of any systems communicating with the systems having the same IP address. If two systems on different IP subnets have the same IP address, then routing packets belonging to the system attempting the impersonation usually fails in the absence of the source routing option.

When the "r" command client uses the source routing option, it is possible that all of the routers along the way, the server network system software, and the "r" command server application software may make use of the source route supplied by the client. It is best to ensure that all of the following conditions are true in order to defeat a trusted host impersonation based on the use of source routing: the router to the subnet on which the server is located does not route packets with the source routing option enabled; the server network system software does not accept a source routed packet; and "r" command server application software checks incoming packets for the source route option and refuses service to such requests.

When a trusted host is not functioning, impersonating that trusted host on the same IP subnet is easily accomplished. It is common for personal computers to be turned off at night. This provides a perfect opportunity to impersonate a personal computer. Workstations are usually left on all the time. If the trusted host is turned off or disconnected from the network, then another system on the same subnet can easily assume the IP address of the trusted host. This situation is not easily detectable.

An "r" command server can employ some measure of protection against impersonation of a trusted host. A daemon, possibly consisting of just a shell script, can be created on the server to periodically monitor the "health" of the server's trusted hosts. Such monitoring could be no more complicated than simply using *ping* to insure that each trusted host is alive. Should a trusted host not respond properly to the monitoring, then entries for that trusted host in the *hosts.equiv* and all *.rhosts* file are removed. When a trusted host comes on line again, it would be required to identify and authenticate itself with the server. This identification and authentication could be done by the trusted host administrator logging into the server or by some automated communication between the trusted host and the server. The entries for the trusted host in the *hosts.equiv* and/or *.rhosts* files would then be replaced.

### 10.2.8 Improving the Security of NFS

There are many threats associated with using NFS. Because of the security problems associated with NFS, NFS should not be run on a secure gateway (see sec. 10.3). This section will discuss precautions which should be taken when using NFS, in particular when exporting files.

#### Exporting Files

When files are exported on an NFS server, the administrator designates which clients can mount specific directory trees located on the server. The type of access may also be given for exported files.

For Sun Microsystems Network Filesystem (NFS), the */etc/exports* file contains entries for directories that can be exported to NFS clients. Each line of the */etc/exports* file has the following format:

*directory -options[,options]...*

"Directory" is the pathname of a directory or a filesystem. "Options" allow a variety of security-related options to be specified. It is important that a system administrator is aware of default access that is allowed if certain options are not specified. It is also important that the system administrator is aware of the implications of using or not using certain options. The following list gives examples of how to export files so that the possibility of unauthorized file access is reduced. Examples given apply to the implementation of NFS in SunOS [SUN90b].

- Files should only be exported to clients that need to use those files. Having a line in the */etc/exports* file of the format "/usr" is strongly discouraged because the /usr directory is being exported to all systems on the network. The "access=client[:client]..." option should be used so that mount access is only given to each client listed. The *client* field can either be a hostname or a netgroup. For remote mounts, the information in a netgroup is used to restrict access to a group of machines.

  The following example exports the /usr directory to clientA and clientB. All other systems are denied permission to mount these files.

  **/usr -access=clientA:clientB**

- Shared files, such as system files, should be exported read only, and owned by root. This will help to prevent system files from being modified. It should be noted that the default is for directories to be exported read-write. The following command exports /usr/bin read-only to the systems clientA, clientB, clientC, and clientD.

  **/usr/bin -ro,-access=clientA:clientB:clientC:clientD**

- Files should not be exported with the "root=client[:client]..." option. This option gives root access for root users from specified clients. If a client is impersonated, then an unauthorized user could modify files on the server.

- When possible, the minimal subdirectory tree should be exported. For example, if access is needed only for /usr/bin, /usr/bin should be exported instead of /usr. It is not possible to export either a parent directory or a subdirectory of an exported directory that is within the same filesystem. For example, it would be illegal to export both /usr and /usr/local if both directories resided on the same disk partition. As a result, sometimes it is necessary to give access to more files for more clients than is desired.

- The *showmount* command can be used to print all directories that are exported for either a local system or a remote system. Systems that do not export directories to specific clients are particularly vulnerable because the output of the *showmount* command will reveal that any client on the network can mount the directory. If a system is using the */etc/hosts* facility, the */etc/exports* file can contain aliases for host names. Using aliases for client names will prevent *showmount* from revealing which clients have permission to mount specific directories.

It is advisable for administrators to regularly inspect the file which gives permission for directories to be exported to clients (i.e., the */etc/exports* file for SunOS) to verify that entries have not been modified.

In order to protect a system from unauthorized setuid and setgid programs, it is advisable to mount all files with the *nosuid* option. Use of the *nosuid* option provides a measure of protection on the client from someone with *root* access on the server gaining *root* access on the client through a setuid program that grants root privileges to the user executing the program. For example, the superuser on the server can create an executable file (e.g., a copy of *sh*) with *setuid root*, i.e., when *sh* is run, it runs as *root*. If this file is exported to a client, then any user on that client who can execute that file can become superuser on the client.

### Protecting Against Impersonation Using NFS

As with the "r" commands, it is often easy to impersonate a user or an NFS client system in an environment where NFS is used. A superuser on a client workstation is able to impersonate any user. With the "r" commands, a user is impersonated by assuming the user's username. With NFS, a user is impersonated by assuming the user's userid.

One way of minimizing the risk of user impersonation with NFS is to only export a user's files to that user's personal computer or workstation. Very often in current network environments, each client system is the exclusive domain of a single user. This type of environment promotes better security with NFS since each NFS client is accessed by only one user and only that user's files need be exported to that user's system. Any other files needed by a user can be exported "read-only" to that user's system. Once an NFS client is able to mount more than one user's files, then the possibility exists for the superuser to impersonate any user on that NFS client and there is no easy way to protect against such impersonation.

If an NFS client is a workstation, then a user is authenticated and associated with a userid by logging into the workstation. If an NFS client is a personal computer, then the NFS client implementation on the personal computer provides some way for the personal computer user to be associated with a userid so that access control can take place on the NFS server. The personal computer user's authentication and association with a userid is sometimes implemented with a daemon called *pcnfsd*. This daemon runs on a server not necessarily an NFS server. The personal computer user is able to designate not only NFS servers from which files are mounted but also the server running *pcnfsd* which authenticates the user to the NFS servers. Herein lies the possibility of a personal computer user assuming the identity of another user. Again, the importance of only exporting a user's files to that user's system is illustrated.

It is possible to impersonate an NFS client in the same manner as impersonating an "r" command trusted host (see sec. 10.2.7). As in the case of the "r" commands, a significant danger here occurs when a legitimate NFS client is disabled, disconnected from the network, or turned off. It is common practice to power off a personal computer at the end of the day. Thus, a personal computer which is also an NFS client can present a problem with regard to impersonation. Server administrators should be aware that almost all client implementations of NFS on personal computers also support the "r" commands. Thus, a personal computer NFS client is almost always a potential "r" command trusted host.

Protection against NFS client impersonation is similar to protection against trusted host impersonation with the "r" commands as described in section 10.2.7. A daemon, which could be a shell script, runs on the NFS server and monitors (perhaps by simply using *ping*) the "health" of each NFS client. When a client does not respond, that client's files on the server are unexported, thus denying access. When the client comes back on line, its files are exported once again after the user is authenticated.

If the NFS client is a personal computer and user authentication is by means of the *pcnfsd* daemon, then *pcnfsd* can be modified (the source is available) to export the user's files when the user is authenticated and receives the userid. If the NFS client is a workstation, then a client command for "pcnfsd" could be readily implemented on the workstation or the user could log into the NFS server to run a command which exports his files.

#### Secure NFS

One of the shortcomings of NFS is that users and systems may be impersonated. Making NFS secure requires assuring that unauthorized users cannot access files stored on secure servers. Secure NFS improves the authentication of NFS requests by using Secure RPC. Secure RPC uses the Data Encryption Standard (DES) and exponential key exchange to verify each NFS RPC request. Section 10.4.6 describes Secure RPC in detail.

### 10.2.9 Improving the Security of RFS

RFS facilitates the sharing of files. Unless precautions are taken when using RFS, unintended access may be granted for shared resources. This is especially true for file systems that are not exported with options specified to control access. The following is a list of ways to make RFS more secure. Commands used pertain to the System V Release 4 version of RFS.

- When starting RFS, issue the command *rfstart -v*. This command will tell RFS to deny connection requests from any system that has not been given a password via the RFS verification procedure. Connection requests will also be denied from any system that specifies an incorrect password. The connection security feature of RFS makes it more difficult for clients to be impersonated.

- Use the -access option on all *share* commands. Hosts not included in the access list will not be permitted to mount the resource.

- Shared files, such as system files, should be exported read only, and owned by root. This will help to prevent system files from being modified.

- For exported file systems, use UNIX file permissions to control access to shared resources.

- Implement user id and group id mappings. This will deter user impersonation attacks. The *idload* command can be used to display current user and group mappings in effect.

- Do not allow untrusted systems to mount file systems with root access enabled.

- The *dfshares* command can be used to display a list of all resources in the domain that are available for mounting via RFS. The *dfmounts* command can be used to display a list of remote hosts that have resources mounted from a server. These commands can be used to assist in monitoring RFS security.

- Do not run RFS on a secure gateway (see sec. 10.3).

### 10.2.10 Improving the Security of NIS

The Network Information Service (NIS) is a distributed database system that simplifies the task of system administration by storing account and configuration files on a single system. NIS is capable of storing a master password file so that users can use the same password for all accounts through out the network. Section 9.2.9 presented a list of known security problems with NIS. The following is a list of remedies for security problems associated with NIS.

- The */etc/hosts.equiv* file should not consist a single line containing a '+' because trusted access should never be granted to all hosts on the network. If the */etc/hosts.equiv* file is used, it should contain entries for specific host names. Specific user names may also be specified.

- Netgroups, which group various users and hosts together, can be defined in the file */etc/netgroup* and maintained as an NIS map. Using netgroups can simplify the task of granting or denying access to users. Netgroups can also be used in the password file.

- When NIS password or group file information is to be accessed, a line of the format '+:' should be used to indicate NIS server access. The format '+::0:0:::' should not be used because if the leading '+' is accidentally deleted, unintended access can be granted.

- NIS map files should be writeable only to the super-user.

- The program *ypbind* should not be started with options that allow *ypbind* to listen to locally-issued *ypset* commands. This prevents an cracker from using *ypset* to obtain information from an unauthorized NIS server. It also prevents a cracker from obtaining unauthorized copies of databases by guessing the name of a NIS domain, binding to the NIS server using the *ypset* command, and requesting a database.

- Password aging can be used to force users to change their passwords periodically. Although password aging cannot be centralized using NIS, password aging can be individually implemented on each system a user can log in to.

- Do not run NIS on a secure gateway (see sec. 10.3).

- To prevent unintended disclosure of information contained in the NIS databases, the *ypserv* program can be modified to only respond to requests from authorized NIS clients. This modification requires access to NIS source code.

- Patches for bugs in *ypserv*, *ypxfrd*, and *portmap* utilities are available from Sun Microsystems.

- A site can migrate from using NIS to NIS+. Security is an integral part of NIS+. When properly configured, NIS+ prevents unauthorized sources from reading, changing, or destroying naming service information [JS92].

## 10.3 Improving Network Security By Means of Secure Gateways (or *Firewalls*)

Local area networking has become a widely used means for organizations to share distributed computing resources. Internet sites often use the TCP/IP protocol suite and UNIX for local area networking purposes, because in addition to providing standard local area network services, UNIX and TCP/IP offer methods for centralizing the management of users and resources. This aids greatly in reducing the amount of work and overhead involved in managing user accounts and making distributed resources available to users. It can also be practical to use the same protocols and services for wide area networking as well as for local area networking.

But, two factors now make using TCP/IP for local area networking an increasingly risky business: a number of the TCP/IP services are inherently flawed and vulnerable to exploitation, and the tremendous growth of the Internet has increased greatly the likelihood of such exploitation. *Crackers* often roam the Internet searching for unprotected sites; misconfigured systems as well as use of insecure protocols make the cracker's job much easier [Bel92]. Two of the TCP/IP services most often used in local area networking, NIS (Network Information Services) and NFS (Network File System), are easily exploited; crackers can use weaknesses in NIS and NFS to read and write files, learn user information, capture passwords, and gain privileged access.

Kerberos and Secure RPC are effective means for reducing risks and vulnerabilities on local area TCP/IP networks, however they suffer from the disadvantages of requiring modified network daemon programs on all participating hosts. For many sites, the most practical method for securing access to systems and use of inherently vulnerable services is to use a Secure Gateway, or *firewall* system. A firewall system resides at an Internet gateway (or any subnet gateway) and blocks certain protocols and services from entering or exiting the protected subnet. A firewall system can also restrict access to hosts, log important network activity, and prevent information about internal systems and users from leaking out to the rest of the Internet.

### 10.3.1 Introduction to Firewalls

As the name implies, a firewall is a protection device to shield vulnerable areas from some form of danger. In the context of the Internet, a firewall is a system, i.e., a router, a personal computer, a host, or a collection of hosts, set up specifically to shield a site or subnet from protocols and services that can be abused from hosts on the outside of the subnet. A firewall system is usually located at a higher-level gateway, such as a site's connection to the Internet, however firewalls can be located at lower-level gateways to provide protection for some smaller collection of hosts or subnets.

The general reasoning behind firewall usage is that without a firewall, a subnet's systems are more exposed to inherently insecure services such as NFS or NIS and to probes and attacks from hosts elsewhere on the network. In a firewall-less environment, network security is totally a function of each host on the network and all hosts must, in a sense, cooperate to achieve a uniformly high level of security. The larger the subnet, the less manageable it is to maintain all hosts at the same level of security. As mistakes and lapses in security become more common, break-ins can occur not as the result of complex attacks, but because of simple errors in configuration and inadequate passwords.

A firewall can greatly improve network security and reduce risks to hosts on the subnet by filtering inherently insecure services and by providing the capability to restrict the types of access to subnet hosts. As a result, the subnet network environment poses fewer risks to hosts, since only selected protocols will be able to pass through the firewall and only selected systems will be able to be accessed from the rest of the network. Eventual errors and configuration problems that reduce host security are better tolerated, as well as the internal use of protocols such as NIS and NFS. A firewall system offers the following specific advantages:

- **concentration of security**, all modified software and logging is located on the firewall system as opposed to being distributed on many hosts;
- **protocol filtering**, where the firewall filters protocols and services that are either not necessary or that cannot be adequately secured from exploitation;
- **information hiding**, in which a firewall can "hide" names of internal systems or electronic mail addresses, thereby revealing less information to outside hosts;
- **application gateways**, where the firewall requires inside or outside users to connect first to the firewall before connecting further, thereby filtering the protocol;
- **extended logging**, in which a firewall can concentrate extended logging of network traffic on one system; and
- **centralized and simplified network services management**, in which services such as ftp, electronic mail, gopher, and other similar services are located on the firewall system(s) as opposed to being maintained on many systems.

A firewall not only filters easily exploited services from entering a subnet, it also permits those services to be used on the inside subnet without fear of exploitation from outside systems. A firewall's protection is bi-directional; it can also protect hosts on the outside of the firewall from attacks originating from hosts on the inside by restricting outbound access.

Given these advantages, there are some disadvantages to using firewalls, the most obvious being that certain types of network access may be hampered or even blocked for some hosts, including telnet, ftp, X Windows, NFS, NIS, etc. However, these disadvantage are not unique to firewalls; network access could be restricted at the host level as well, depending on a site's security policy.

A second disadvantage with a firewall system is that it concentrates security in one spot as opposed to distributing it among systems, thus a compromise of the firewall could be disastrous to other less-protected systems on the subnet. This weakness can be countered, however, with the argument that lapses and weaknesses in security are more likely to be found as the number of systems in a subnet increase, thereby multiplying the ways in which subnets can be exploited.

Another disadvantage is that relatively few vendors have offered firewall systems until very recently. Most firewalls have been somewhat "hand-built" by site administrators, however the time and effort that could go into constructing a firewall may outweigh the cost of a vendor solution. There is also no firm definition of what constitutes a firewall; the term "firewall" can mean many things to many people.

### 10.3.2 Firewall Components

There are three primary components (or aspects) for firewall systems, those being

- **packet filtering**
- **application gateways**
- **logging and detection of suspicious activity**

The last item may range in capability, from creating log entries for excessive login attempts to notification of operators via e-mail or pagers to intrusion/detection systems that build user profiles and raise alarms when out-of-bound behavior occurs.

Up until now, the term "firewall" has been used here somewhat loosely, since firewall systems can range greatly in how well they implement the above components. The most common type of firewall is simply a router that has the capability to filter TCP/IP packets based on information fields in each packet. Less common but more secure are systems that include packet filtering as well as logging and application gateways for telnet, ftp, or e-mail. These firewalls may actually be a collection of systems such as a router, an application gateway system, and a system for logging. Also found are firewall systems that simply block *all* traffic, thus completely cutting off network access except for those users with accounts on the firewall system. However, since packet filtering capability appears to be the common component in most firewall systems, the following paragraphs go into more detail on packet filtering than the other components.

### Packet Filtering

The primary activity of a firewall is filtering packets that pass to and from the Internet and the protected subnet. Filtering packets can limit or disable services such as NFS or telnet, restrict access to and from specific systems or domains, and hide information about subnets. A firewall could filter the following fields within packets:

- **packet type**, such as IP, UDP, ICMP, or TCP;
- **source IP address**, the system from which the packet originated;
- **destination IP address**, the system for which the packet is destined;
- **destination TCP/UDP port**, a number designating a *service* such as telnet, ftp, smtp, nfs, etc., located on the destination host, and
- **source TCP/UDP port**, the port number of the service on the host originating the connection.

In almost all cases, packet filtering is done using a *packet filtering router* designed for filtering packets as they pass between the router's interfaces. Packet filtering capability is usually not included in operating systems such as UNIX or VAX/VMS, however at least one vendor includes packet filtering capability [Ran92]. Not all packet filtering routers can filter based on source TCP/UDP port, however more vendors are starting to incorporate this capability.

### Which Protocols to Filter

The decision to filter certain protocols and fields depends on the site security policy, i.e., which systems should have Internet access and the type of access to permit. The location of the firewall will influence the policy. For example, if the firewall is located at a site's Internet gateway, the decision to block inbound telnet access will still permit site systems to access other site systems. If the firewall is located at a subnet within the site, the decision to block inbound telnet to the subnet will prevent access from other site subnets.

Usually, the following services are **blocked** at a firewall [Bel89]:

- **tftp**, trivial ftp, used for booting diskless workstations, terminal servers and routers, can also be used to read any file on the system if set up incorrectly;
- **X Windows, Sun OpenWindows**, can leak information from X window displays, such as all keystrokes;
- **SunRPC**, including NIS and NFS, which can be used to steal system information such as passwords and read and write to files; and
- **rlogin, rsh, rexec, and other "r" services**, services that if improperly configured can permit unauthorized access to accounts and commands.

These services are inherently open to abuse and therefore should be blocked directly at the firewall. Other services, whether inherently dangerous or not, are usually filtered and possibly restricted to only those systems that need them. These would include:

- **telnet**, often restricted to only certain systems;
- **ftp**, like telnet, often restricted to only certain systems;
- **SMTP**, often restricted to a central e-mail server;
- **RIP**, routing information protocol, which can be spoofed to redirect packet routing to the wrong place causing denial of service on the network, is often unnecessary if a single default route exists;
- **DNS**, domain names service zone transfers, contains names of hosts and information about hosts that could be helpful to attackers;
- **UUCP**, Unix-Unix Copy Protocol, if improperly configured can be used for unauthorized access;
- **NNTP**, network news transfer protocol, to enable reading and transfer of Usenet news groups;
- **NTP**, network time protocol, for synchronizing system clocks according to the atomic clock maintained by the U.S. Dept. of Commerce.

While some of these services such as telnet or ftp are inherently risky, blocking these services completely may be too draconian a policy for many sites. Not all systems, though, generally require access to all services. For example, restricting telnet or ftp access *from* the Internet to only those systems that require the access can improve security at no cost to user convenience. Services such as NNTP or NTP may seem to pose no threat, however restricting these protocols to only those systems that need them helps to create a "cleaner" network environment and reduces the likelihood of exploitation from yet-to-be-discovered vulnerabilities and threats.

UNIX systems, including System V and BSD-based, generally contain TCP/IP code or conventions derived from the original Berkeley UNIX distributions. The TCP/UDP port allocation scheme used by Berkeley is therefore common to most UNIX systems as well as most other non-UNIX TCP/IP implementations. This informal standard aids greatly in packet filtering schemes.

In the Berkeley scheme, port numbers between 0 and 1023 are *privileged*, that is, these ports are used to connect to services such as telnet, ftp, and SMTP daemons that require system-level privileges. Port numbers above 1023 are *usually* associated with processes that don't need special privileges.

An example may best illustrate the port allocation scheme. If a user on a remote system wishes to telnet to a local system, the remote system's telnet client allocates, on behalf of the user, an unprivileged port with number greater than 1023 (say, 2123). It then sends an initial packet containing port number 2123 to the telnet server's port on the local system. The telnet server, a privileged process, resides at port number 23. The local system, in accepting the telnet connection, would then respond back to the telnet client on the remote system by sending a packet to the client's port number 2123. The connection is thus established, and the client and server can proceed with exchanging data and keystrokes. Most other privileged services follow this scheme, with some exceptions, thus filtering on privileged port numbers can greatly simplify packet-filtering rules.

To illustrate how the Berkeley port allocation scheme can simplify packet filtering, a site that wishes to block inbound TCP/IP services could simply block all packets from outside systems with port numbers less than 1024. This, of course, would block all telnet, ftp, SMTP, and most other services from entering the site. To allow specific services to pass through, then, one can make exemptions for specific ports.

Although this scheme is relatively simple, there are a number of problems associated with it, the primary one being that not all privileged services use privileged ports. X Windows, Sun OpenWindows, and some ports allocated by the portmapper process all can be greater than 1023, thus filtering on specific port numbers above 1023 is still required to block these protocols. Another problem is that the Berkeley port allocation scheme is only a commonly-used convention, not a true standard. As a result, systems that do not follow this scheme may be able to evade packet-filtering rules. Fortunately, deviations from the scheme are relatively rare. The following section provides examples in more detail for filtering packets.

### Examples of Packet Filtering

Using the privileged ports convention, this section contains several examples of packet filtering rules. In the examples, the syntax *a.b.c.d/y* denotes the 32-bit IP address *a.b.c.d* with the left-most *y* bits of the address significant for a comparison, as used in [Cha92]. For example, 129.7.0.0/16 means that the first 16 bits, 129.7, are significant for comparisons to other addresses or patterns. Thus, 129.7.3.8 matches 129.0.0.0/8, 129.7.0.0/16, and 129.7.3.0/24, but not 129.7.4/24. An address or pattern with 0 significant bits such as 0.0.0.0/0 matches *any* address, while a pattern with 32 significant bits such as 129.7.3.8/32 matches only that specific address.

The following examples assume packet-filtering routers (or dual-homed hosts with packet filtering capability) with *two* interfaces (Ethernet, token ring, etc.).

The first example is a simple method to block access to all privileged ports from outside systems to a protected network:

| Type | Source Addr | Dest Addr | Source Port | Dest Port | Action |
|---|---|---|---|---|---|
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | * | $< 1024$ | deny |

This effectively blocks all access from the outside to the protected network, however systems on the protected network may still be able to use telnet, ftp, and some other services to connect outbound.

This example may prove too restrictive for many sites, so the next example preserves outbound access from the protected network to the outside and permits only inbound telnet (port 23), ftp (ports 20 and 21), and SMTP (port 25) access from outside systems to the protected network. The following rules would filter accordingly:

| Type | Source Addr | Dest Addr | Source Port | Dest Port | Action |
|---|---|---|---|---|---|
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | 20 | permit |
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | 21 | permit |
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | 23 | permit |
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | 25 | permit |
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | $< 1024$ | deny |

However, to make this example more complete, we would need to block inbound access to those services that use port numbers above 1023, such as X Windows (ports 6000, 6001, up to 60*nn*, where *nn* is the maximum number of X displays running on any one host) and Sun OpenWindows (2000). The following rules would need to be added:

| Type | Source Addr | Dest Addr | Source Port | Dest Port | Action |
|---|---|---|---|---|---|
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | 2000 | deny |
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | 6000 | deny |
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | 6001 | deny |
| tcp | 0.0.0.0/0 | 0.0.0.0/0 | $*$ | 6002 | deny |

The above examples have all used port numbers or packet type as the filtering criteria. Source and destination IP addresses combined with the other header fields can permit certain types of access to occur only to designated systems or subnets. For example, a site may wish to allow certain services from the outside such as SMTP, ftp, or NNTP (port 119), to go to only specific systems. In the following example, one host on the protected network, 127.32.7.20, is acting as the site's anonymous ftp server, a second host, 127.32.7.21, is the e-mail server, and a third host, 127.32.7.22, is the news server. The rules for limiting inbound access from the outside to these systems would be as follows:

| Type | Source Addr | Dest Addr | Source Port | Dest Port | Action |
|---|---|---|---|---|---|
| tcp | 0.0.0.0/0 | 127.32.7.20/32 | $*$ | 20 | permit |
| tcp | 0.0.0.0/0 | 127.32.7.20/32 | $*$ | 21 | permit |
| tcp | 0.0.0.0/0 | 127.32.7.21/32 | $*$ | 25 | permit |
| tcp | 0.0.0.0/0 | 127.32.7.22/32 | $*$ | 119 | permit |

For more detailed examples of packet filtering, refer to [SQ92].

### Alternatives to Packet Filtering

In the absence of packet filtering capability, there are several alternatives, however none of them are as flexible or powerful as a packet-filtering router or host. [GS91] describes a method by which a *dual-homed host*, that is, a host with two interfaces used as a subnet gateway, can be used to block all TCP/IP traffic from entering or leaving the protected subnet. IP forwarding would be disabled at the host, and any users who might wish to telnet or otherwise access outside systems would log into the gateway itself. This arrangement is somewhat restrictive since it requires users to connect to the gateway before connecting inward or outward, however can be very secure and more cost-effective for small sites.

Another alternative is for all hosts to use third-party packages that provide access control to certain services. [Ven92] has created a "TCP Wrapper" package that is available via anonymous ftp and serves as a front-end to all services executed from the UNIX inetd daemon process, which include telnet, ftp, "r" services, and possibly SMTP. The front-end checks to determine whether the host requesting the connection is permitted and then either accepts or rejects the connection. The requesting host's address can be matched against a pattern.

The TCP Wrapper package does not protect other UDP-based services such as NIS, NFS, DNS, and so forth that are not invoked via the inetd daemon process. [LeF92] has created a package called "Securelib" for SunOS systems that can be used to provide access control to services mapped by the portmapper process. Using a similar method of pattern matching against the requesting host's address, a host can deny or accept requests to the portmapper. However, since the portmapper can be bypassed by determined crackers, this method does not provide the same degree of protection as does true packet-filtering capability. At the same time, the TCP Wrapper and Securelib packages provide a much higher level of security than default levels and would block casual attempts to exploit protocols.

### Logging and Detection of Suspicious Activity

Packet-filtering routers unfortunately suffer from a number of weaknesses. The filtering rules can be difficult to specify, usually no testing facility exists thus testing must be done manually, and the filtering rules can be very complex depending on the site's access requirements. No logging capability exists, thus if a router's rules still let "dangerous" packets through, the packets may not be detected until a break-in has occurred. In addition, some packet filtering routers filter only on the destination address not on the source address.

Some logging capability within a firewall system is important to ensure the secure operation of the firewall and to detect suspicious activity that might lead to break-ins. A host system with packet-filtering capability such as [Ran92] or [Rap93] can more readily monitor traffic than, say, a host in combination with a packet-filtering router, unless the router can be configured to send all rejected packets to a logging host.

What type of traffic should be logged? In addition to standard logging that would include statistics on packet types, frequency, and source/destination addresses, the following types of activity should be captured:

- **connection information**, including point of origin, destination, username, time of day, and duration;
- **attempts to use any "banned" protocols** such as tftp, domain name service zone transfers, portmapper and RPC-based services, all of which would be indicative of probing or attempts to break in;
- **attempts to spoof internal systems** such as traffic from outside systems attempting to masquerade as internal system; and
- **routing re-directions** that come from unauthorized sources (unknown routers).

Logs will have to be read frequently. If suspicious behavior is detected, a call to the site's administrator can often determine the source of the behavior and put an end to it, however the firewall administrator also has the option of blocking traffic from the offending site. [GS91] and [PR91] contain useful advice on dealing with suspicious activity and break-ins.

### Application Gateways

After packet filtering and logging, application gateways function to provide a higher level of security for applications such as telnet, ftp, or SMTP that are not blocked at the firewall. An application gateway is typically located such that all application traffic destined for hosts within the protected subnet must first be sent to the application gateway (in other words, any application traffic that is not directed at the application gateway gets rejected via packet filtering). After performing some action, the application gateway may pass the traffic on to a host or may reject the traffic if it is not authorized. Application gateways are also referred to as "proxy servers."

A site would use application gateways to provide a "guarded gate" through which application traffic must first pass before being permitted access to specific systems. As an example of an application gateway for telnet, a site might advertise only the name of the telnet gateway to outside users and not the names of specific hosts. The protocol for connecting to specific internal hosts would be as follows:

1. a user first telnets to the application gateway and enters the name of the desired host;
2. the gateway perhaps checks the user's source IP address and accepts or rejects it according to any access criteria in place;
3. the user may need to authenticate herself using an authentication token such as a challenge-response device;
4. the gateway then creates a telnet connection to the desired host;
5. the user's system knows only that the telnet session is between the user's system and the application gateway; and

6. the application gateway logs the connection, including the connection's origination address, destination, time of day, and duration.

Application gateways, then, have a number of advantages over the default mode of permitting application traffic directly to internal hosts:

- **information hiding**, in which the names of internal systems need not necessarily be made known via DNS to outside systems, since the application gateway may be the only host whose name must be made known to outside systems;

- **robust authentication and logging**, in which the application traffic can be pre-authenticated before it reaches internal hosts and can be logged more effectively than if logged with standard host logging;

- **cost-effectiveness**; because third-party software or hardware for authentication or logging need be located only at the application gateway; and

- **less-complex filtering rules**, in which the rules at the packet-filtering router will be less complex than they would if the router needed to filter application traffic and direct it to a number of specific systems. The router need only allow application traffic destined for the application gateway and reject the rest.

A disadvantage of application gateways is that, in the case of client-server protocols such as telnet, two steps are required to connect inbound or outbound. This may prove somewhat tedious for users, however it is a small price to pay for the increase in security.

Application gateways are used generally for telnet, ftp, and e-mail. [Ran92] uses one application gateway for both telnet and ftp, and another for e-mail. The telnet application works as described in the earlier example; the ftp application includes the capability to deny *puts* and *gets* to specific hosts as required. For example, an outside user who has established a ftp session (via the ftp application gateway) to an internal system such as an anonymous ftp server might try to upload files to the server. The application gateway can filter the ftp protocol and deny all *puts* to the anonymous ftp server; this would ensure that nothing can be uploaded to the server and would provide a higher degree of assurance than relying only on file permissions at the anonymous ftp server to be set correctly.

An e-mail application gateway serves to centralize e-mail collection and distribution to internal hosts and users. To outside users, all internal users would have e-mail addresses of the form:

*user@emailhost.b.c.d*

where *emailhost* is the name of the e-mail gateway. The gateway would accept mail from outside users and then forward mail along to other internal systems as necessary, using *aliases* or forward files. Users sending e-mail from internal systems could send it directly from their hosts, or in the case where internal system names are not known outside the protected subnet, the mail would be sent to the application gateway, which could then forward the mail to the destination host.

Application gateways are also ideal locations for services such as anonymous ftp, gopher, and other information distribution servers. Both [GS91] and [Che90] go into more detail on setting up application servers. [Ran92] and [Ran93] discuss location of application servers and filtering rules for directing application traffic to application gateways.

### Examples of Firewalls

Now that the basic components of firewalls have been examined, some examples of different firewall configurations will give readers a more concrete understanding of firewall implementation. The firewall examples shown here are:

- **packet-filtering-only firewall**;
- **dual-homed gateway**;
- **choke-gate firewall**; and
- **screened-subnet firewall**.

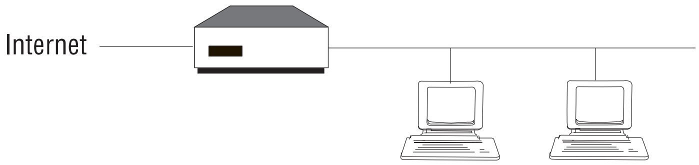



Figure 10.1: Packet-filtering-only firewall.

The packet-filtering-only firewall (fig. 10.1) is perhaps most common and easiest to employ. Basically, one installs a packet-filtering router at the Internet (or any subnet) gateway and then configures the packet-filtering rules in the router to block or filter protocols and addresses. The systems "behind" the router usually have direct access to the Internet, however inherently-dangerous services such as NIS, NFS, and X Windows are usually blocked.

Depending on the flexibility of the filtering rules as well as the size of the protected subnet, the packet-filtering-only firewall may be adequate for many sites. However, there are a number of disadvantages with this approach, including the following:

- packet-filtering routers generally do not provide logging capability, thus an administrator may not easily determine whether the router has been compromised or is under attack;
- packet-filtering rules are often difficult to configure and test thoroughly; and
- if complex-filtering rules are required or if there is a large number of hosts, the filtering rules may become unmanageable.

Thus, a packet-filtering-only firewall is best suited to environments that do not require complex filtering or that do not have a large number of hosts to protect. Sites with high security needs may wish to consider a more robust firewall such as the filter-choke or screened-subnet firewall.

The dual-homed gateway (fig. 10.2) is used as an alternative to packet-filtering routers. The gateway host system is configured to block all traffic between the Internet and the protected subnet by disabling IP forwarding capability. Users on internal systems can gain access to Internet systems by either having accounts on the gateway itself, or by configuring the gateway to pass certain protocols such as telnet, ftp, or mail (i.e., an application gateway).


Figure 10.2: Dual-homed gateway.

Dual-homed gateways are often the least-expensive option for many sites and, if used mainly as an application gateway, can be quite secure. Unlike packet-filtering routers, a dual-homed gateway can perform some logging and provide more evidence to administrators of attacks or break-ins. Unfortunately, configuring the gateway to act as an application gateway can require modified operating system software. In situations in which modified software is not possible, users need to log on to the gateway to access the Internet. This may present a problem if there are a large number of users, since either every user must have an account on the gateway or group accounts must be used. If a user's account is somehow compromised, the intruder could potentially subvert the firewall and re-enable IP routing. Authentication tokens, Kerberos, and other methods should be used to decrease the likelihood of break-ins.

[GS91] discusses a method to pass ftp and telnet traffic that uses group accounts in a creative way. Essentially, a group account for telnet, called *telnetout* is created along with a .rhosts file that lists all the internal users who are allowed to telnet out to Internet hosts. Users can then rlogin to the gateway without requiring individual accounts, and the .rhosts file restricts which users and systems can login much better than a wide-open group account. The ftp service is configured the same way, with a *ftpout* account that users can rlogin to and then use the ftp service on the gateway to transfer files with Internet hosts. Of course, security on the gateway must be quite high, since any compromise of the telnet and ftp accounts could wreak havoc. Other user accounts on the gateway should be kept to a minimum.

The dual-homed gateway must be set up to pass e-mail to and from internal systems. For mail destined to internal systems, simple mail aliases can be used at the gateway to forward mail. For mail from internal to outside systems, the mailers on internal systems must be configured so that all mail not destined for internal systems is sent to the gateway. The gateway would then rewrite the message headers and forward the mail on to the outside system. Both [GS91] and [Ran93] discuss advantages and disadvantages of dual-homed gateways used as firewalls.

The choke-gate gateway (fig. 10.3) is a step up in terms of security and flexibility from the filtering-only and dual-homed firewalls. It combines a packet-filtering router with an application gateway located on the internal side of the router ([GS91] refers to the application gateway as the *gate* and the router as the *choke*). The application gateway is used for passing telnet, ftp, and SMTP. The router filters or blocks inherently dangerous protocols, however it also rejects (or accepts) application traffic according to the following:

- for traffic originating from outside systems, any application traffic sent to the application gateway is passed along;
- traffic originating from the outside to any internal system *but* the application gateway is rejected; and
- the router rejects any application traffic originating from the inside except traffic from the application gateway.

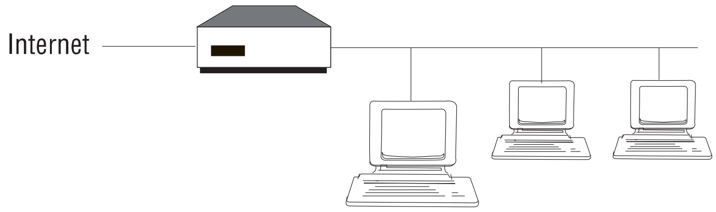



Figure 10.3: Choke-gate firewall.

The gate would be logically set up like the dual-homed firewall to forward e-mail, and would handle ftp and telnet traffic using group accounts and .rhost files. Note that figure 10.3 shows the gate physically connected to the same subnet as other systems behind the choke. The choke-gate firewall is more flexible than the dual-homed firewall, however, and more secure. Unlike the dual-homed gateway, the gate does not need to block all IP traffic; less-risky traffic such as NTP, NNTP, or SMTP can be restricted to certain internal systems via the packet-filtering router. Both the choke and the gate would need to be compromised to fully subvert the firewall. Refer to [GS91] for more details on setting up a choke-gate firewall.

Some vendors have offered gateway products that appear as *hybrid* dual-homed gateways. The products may use modified operating system software to filter packets and pass protocols such as telnet and ftp. [Ran93] and [Ran92] discuss one such firewall that uses separate systems for application gateways and a "screened" subnet between the Internet and the internal subnet to isolate one of the application gateways. In figure 10.4, a router is shown as the connection point to the Internet; the router would be used as well to block packets such as NFS, NIS, or any other protocols that should not be allowed to pass to or from the Internet. On the screened subnet, a telnet/ftp application gateway is used for all telnet and ftp traffic. A dual-home gateway with packet-filtering capability passes traffic between the internal subnet and the Internet. An e-mail application gateway resides on the internal subnet; all e-mail to internal systems must be sent to the e-mail gateway.

The dual-homed gateway acts as a second packet-filter, however it enforces the following:

- no telnet or ftp traffic from the outside is allowed to pass to the internal subnet unless it comes from the telnet/ftp gateway;

- no SMTP traffic can pass unless destined for the e-mail gateway; and
- other protocols are restricted as desired.

Depending on site policies, all ftp or telnet traffic from internal systems may be forced to use the telnet/ftp gateway and similarly with e-mail. The dual-homed gateway would in essence trust traffic only from or to the application gateways.

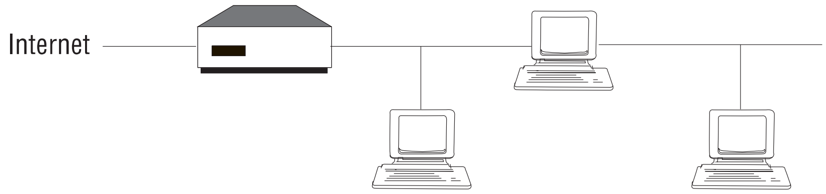



Figure 10.4: Screened subnet firewall.

The telnet/ftp and e-mail gateways could be set up such that they would be the only systems accessible from the Internet; no other system name need be known or used in a DNS database that would be accessible to outside systems. The telnet/ftp application gateways act as proxies: users from the outside (or possibly the inside) would need to connect first to the gateway, authenticate themselves using possibly an authentication token, and then connect internally as permitted. The ftp gateway filters the ftp protocol itself, with the capability to deny *puts* or *gets* to or from specific systems.

This type of firewall arrangement provides a high level of security and offers more flexibility for internal systems that need to connect to the Internet. It is, of course, more complex to configure, however the use of separate hosts for application gateways and packet filters keeps the configuration more simple and manageable. Refer to [Ran93] and [Ran92] for more details on screened-subnet firewalls.

### 10.3.3 Special Considerations With Firewalls

Because the compromise of a firewall would be potentially disastrous to subnet security, a number of special considerations need to be taken with regard to firewall configuration and use. The following list, adapted from [GS91], summarizes these items:

- limit firewall accounts to only those absolutely necessary, such as for the administrator. If practical, disable network logins.
- use authentication tokens to provide a much higher degree of security than that provided by simple passwords. Challenge-response and one-time password cards are easily integrated with most popular operating systems.
- remove compilers, editors, and other program-development tools from the firewall system(s) that could enable a cracker to install Trojan horse software or backdoors.

- do not run any vulnerable protocols on the firewall such as tftp, NIS, NFS, UUCP, or X.

- the finger protocol can leak valuable user information, consider disabling finger.

- on e-mail gateways, consider disabling the EXPN and VRFY commands, which can be used by crackers to probe for user addresses.

- do not permit the firewall systems to "trust" other systems; the firewall should not be equivalent to any other system.

- disable any feature of the firewall system that is not needed, including other network access, user shells, applications, and so forth.

- turn on full-logging at the firewall and read the logs routinely.

### 10.3.4 The Role of Security Policy in Firewall Administration

Lastly, the role of site security policy is especially important with regard to firewall administration. A firewall should be viewed as an implementation of a policy; policy should never be made by the firewall implementation. In other words, agreement on what protocols to filter, application gateways, and other items regarding the nature of network connectivity need to be codified *beforehand*, because ad hoc decisions will be difficult to defend and will eventually complicate firewall administration.

As an example of the above, suppose a firewall is installed that blocks RPC-based traffic from entering or leaving a protected subnet. Later, users on hosts within the subnet wish to use RPC services between hosts on the outside. If no policy exists to defend the RPC filtering rules, it may be difficult to deny access to the hosts, especially if productivity would be impaired by continuing to enforce the filtering. Once exceptions are made, they will most likely continue to be made, until the level of filtering becomes very weak, or the filtering rules become so complex as to be unmanageable.

The example points out that filtering and connectivity policy needs to incorporate not only security needs, but the computing needs of the organization. If the computing needs are ignored or short-changed, the firewall may become too complex to administer or may become essentially useless. Security requirements need to be weighed carefully and accommodations may need to be made if productivity will be hampered by the security policy. In some cases, moving a firewall "higher up" in a subnet, such as locating it at a site's Internet gateway as opposed to a subnet, will solve many problems. For more information, [PR91] contains useful advice on creating security policies for Internet sites that incorporate modes of work and network connectivity requirements.

## 10.4 Robust Authentication Procedures

### 10.4.1 Identification and Authentication

With few exceptions, there is a need in modern network environments to:

1. control access to the network itself.
2. control access to the resources and services provided by the network.
3. be able to verify that the mechanisms used to control that access are providing proper protection.

Controlling access to the network is provided by the network's identification and authentication service. This service is pivotal in providing for (2) and (3) above. If network users are not properly identified, and if that identification is not proven through authentication, there can be no trust that access to network resources and services is being properly controlled and executed.

Authentication is the verification of the entity's identification. That is the host, to whom the entity must prove his identity, trusts (through an authentication process) that the entity is in fact who he claims to be. The threat to the network that the identification and authentication service must protect against is impersonation. According to [TA91], impersonation can be achieved by:

- **forgery**, attempting to guess or otherwise fabricate the evidence that the impersonator knows or possesses the authenticating information (the secret);
- **replay**, where one can eavesdrop upon another's authentication exchange and learn enough to impersonate a user; and
- **interception**, where one is able to slip in-between the communications and "hijack" the communications channel.

### 10.4.2 Distributed System Authentication

According to [WL92], there are three main types of authentication in a distributed computing system.

- Message Content Authentication: the ability to verify that the message received is exactly the message that was sent. Message Content Authentication can be achieved by:
  1. applying a cryptographic checksum called a message authentication code (MAC), or
  2. by applying a public-key digital signature.

The National Institute of Standards and Technology (NIST) Federal Information Processing Standard Publication (FIPS PUB) 113, "Computer Data Authentication" [FIP85] provides information on the use of NIST approved Message Authentication Code Standard, while the Draft FIPS PUB "Digital Signature Standard" [FIP93c] describes the NIST proposed digital signature standard.

- Message Origin Authentication: The ability to verify that the actual sender of a received message is in fact the sender claimed in the message. Using a symmetric (secret key) cryptosystem, the receiver of a message can be assured of the validity of the sender since only the sender and receiver of the message possess the key used to encrypt the message. This type of system needs a trusted third party, however, to provide a non-repudiation service. In an asymmetric (public key) cryptosystem, the use of a public key or digital signature can provide message origin authentication.

- General Identity Authentication: the ability to verify that a principal's identity is who is claimed. It is this type of authentication that is the focus of Section 10.4. The other two types of authentication, message content authentication and message origin authentication will be discussed when they are coupled with identity authentication in the authentication systems that will be examined.

### 10.4.3 The Need: Identity Authentication

The most common authentication model that has been used and is still primarily supplied by operating system manufacturers is the password authentication model. A user supplies a password to the host in order to be authenticated. The host then usually performs a oneway function on the password, and compares the result to the value the host has stored and associates with the user. If the two match, the host trusts that the user is who is claimed.

This model served its purpose for standalone systems, in relatively benign environments where the user supplied password traveled only a short distance, directly from the user terminal to the host. Physical security solutions in these environments were also highly developed. Vulnerabilities that exist with this model in a standalone environment include:

1. the password being sent in plaintext to the host.

2. user-generated passwords being relatively easy to guess.

The use of passwords in the environment mentioned above provided adequate authentication. However in the highly networked, distributed environments currently in use, this model does not fare so well in providing robust, trustworthy authentication. These networked environments are the targets of more hostile threats, more sophisticated attackers, and it could be argued, much more damaging consequences.

Users are utilizing many machines, not just by remotely logging in, but transparently using the services provided on the remote machines. The client/server model allows users to remotely and transparently access process time, files, printers, etc. In these environments, the need for authentication extends far beyond that of supplying a simple password to a local machine.

Authentication of the user to the remote host (or service) and also authentication of the remote host (or service) to the user requires much more sophisticated techniques than that of a simple password. Passwords should not be sent in cleartext over a network. Passwords should be properly managed in a network with many machines, where each user can have a unique password for each machine. Accessing resources on a remote machine requires "virtual identification" for each read/write file access request. Transmitting a simple password each time creates vulnerabilities that extremely limit the effectiveness of the password. And finally, after years and years of user guidance on the importance of choosing robust passwords, users are still generating and using easy-to-guess passwords. For systems (and even closed networks) that contain no sensitive information, this may be acceptable, although these systems and networks may be hard to find today. The sensitivity of the information processed on modern networks, along with the critical functions performed on them, demand that more robust authentication techniques be used. These robust techniques are referred to as strong authentication techniques.

Tardo [TA91] refers to strong authentication as "techniques that permit entities to provide evidence that they know a particular secret without revealing the secret." Strong authentication does not provide to the authenticator, nor to an eavesdropper, any information that could allow them to impersonate (at any time) the entity being authenticated. Traditional password authentication mechanisms, used in a network, are not strong authentication mechanisms because they usually involve transmitting the password over a medium in cleartext and because they are usually received by the authenticator in a form that may be captured, stored and used later for impersonation.

The authentication systems that will be discussed here can be considered strong authentication systems. They rely on the use of cryptographic techniques to provide as little, or no information that could be used for impersonation. Both systems utilize the Data Encryption Standard (DES) to protect authenticating information as it travels through the network. The authentication systems that will be examined are:

- Kerberos - "The Kerberos Authentication Service was originally developed at the Massachusetts Institute of Technology (MIT) for its own use to protect Project Athena's emerging network services. Versions 1 through 3 were internal development versions; protocol version 4 has achieved widespread use. Protocol version 5 incorporates new features suggested by experience with version 4 which make it useful in more situations." [Koh91]

- Secure RPC - "In the mid-1980s, Sun Microsystems developed its own system for improving network security called *Secure RPC*, which was first released with the SunOS 4.0 operating system. Secure RPC is similar to Kerberos, in that it uses the DES to pass confidential information over the network." [GS91]

### 10.4.4 Properties of Distributed Athentication Systems

All authentication systems have some common properties. The properties compiled here have been chosen because they pertain more to the networking aspects of these authentication systems than do some others.

#### The Protocol Used to Verify the Authentication

There are three types of accesses or logins that can be discussed in a network. The first is the local login. This is where the user authenticates himself to the local system (called the client here), usually by supplying a password (although interest in using smartcards/tokens is growing). The second type of login is the remote login. This is when the user from a local system logs into a remote system. For example a user might use a telnet service to login to a remote system. The third type of access that requires authentication is a client/server request. An example of this is when a user mounts on his local machine a remote file system and makes requests to access those files. The protocols used for each of these accesses will be examined for both Kerberos and Secure RPC.

Woo [WL92] defines a protocol as a "precisely defined sequence of communication and computation steps. A communication step transfers messages from one principal (sender) to another principal (receiver), while a computation step updates a principal's internal state. Two distinct states can be identified upon protocol termination, one signifying successful authentication and the other failure". The following format is used to describe the protocols for each system. A communication step "U $\rightarrow$ H : username" defines that a user (U) sends to a host (H) a password. A computation step "H: compute oneway(password)" defines that a host computes a one-way function of a password.

#### The Principals

These systems are used to authenticate a user to a server or service. Each system defines the following common set of principals. To ease the comparison of these systems, the following names will be used to define the common principals:

Username (U) - The identity of a user.
Hostname (H) - The identity of a host.
Client (C) - A host, acting on behalf of a user, requesting services of a host.
Server (S) - A host who provides services.
Certification Authorities (CA) - A server that is used to provide the authentication information. (This may not reside on a dedicated server.)

#### The Areas of the Network Where Trust is Placed

Some part of the network must provide the information that is used to authenticate the user. This can be a special server (sometimes referred to as an authentication server or key distribution server) and contains a database that is used to hold public, private or secret keys of users, clients and servers. Users, clients and servers must trust that the information they receive from this database is correct. Although the actual authentication verification rests with the server that has the requested service, it must trust that the ticket/certificate or other authentication information presented to it came from the authentication server.

### The Areas of the Network Where Secrets are Kept

In these authentication systems, the secrets (such as keys and passwords) are stored in some part of a host (either client, server or both). This implies that the general system-supplied protection mechanisms are used to protect the secrets. An interesting aspect to consider is the mechanisms that are used to protect this information. While there is discussion from some of the authors who discuss these systems regarding the use of smartcards for secure storage, most of them do not view smartcard usage as necessary. See [NIS91a], [Koh91], and [Lin90] for more discussion on the use of smartcards.

### The Key Generation and Distribution Models Used

All of the authentication systems examined use cryptography for authentication. This aspect looks at the models and procedures used to generate and distribute the keys used for the authentication process. This section does not discuss the generation or distribution of the users' secret keys, public key/private key pair, or the use of key encrypting keys. This section focuses on how the session key, or secret key used between client and server are generated and distributed. Session key generation is a major difference between Kerberos and Secure RPC. Kerberos generates a pseudo-random DES key for use as a session key, while Secure RPC uses a public key generation method.

### The Composition of the Ticket/Certificate

The information used to authenticate principals in these systems are comprised in some form of a ticket or certificate. These tickets (also called credentials) contain ids, keys and other pieces of information that are used to provide identities. These tickets alone do not verify authentication. Accompanying these tickets is some form of an authenticator or verifier that, used in conjunction with the ticket, verifies the identity by providing a time reference of usage. Since tickets have distinct lifetimes, it is assumed that the user that has the credential and that is presenting that credential within the appropriate time-frame is the named user. Tickets expire to prevent pre or post-usage. A ticket that is compromised by an intruder can only be used by the intruder through the lifetime of the ticket (from the conclusion of the session or the expiration of the ticket). In these systems, the user can set the lifetime of a ticket up to a specified maximum. Tickets that provide authentication are usually created by a trusted principal. The format used to describe the contents of these tickets is as follows:

| | |
|---|---|
| Password (PW) | The password of a principal. |
| Public key (PB) | The world known key of a principal's public/private key pair. |
| Private key (PR) | The key, of the public/private key pair not shared with anyone, known only to the user. |
| Secret key (SK) | A key, with a relatively long life shared between two principals. |
| Conversation key (CK) | A key, usually used during the lifetime of a session, or a set number of hours, shared between two principals. |

Properties that are beyond the scope of this paper and will not be discussed include:

1. Cascading delegation of authentication.

2. Uncommon security problem controls (servers crashing etc.).

### 10.4.5 Kerberos

Kohl [Koh91] describes the Kerberos model as follows: "Kerberos was developed to enable network applications to securely identify their peers. To achieve this, the initiating party (the client) conducts a three-party message exchange in order to send the contacted party (the server) an assurance of the client's identity. This assurance takes the form of a ticket, which identifies the client, and an authenticator which serves to validate the use of that ticket and prevent an intruder from replaying the same ticket to the server in a future session. A ticket is only valid for a given interval, called a lifetime. When the interval ends, the ticket expires; any later authentication exchanges would require a new ticket."

Kerberos can be used for local logins, remote authentication, and for client/server requests. Where applicable, differences between Kerberos version 4 and Kerberos version 5 are pointed out.

Message Control Authentication can be ensured through the use of the session key between the client and server using the CBC mode of DES. Message Origin Authentication is provided through the use of the protocols verifying the General Identity Authentication.

#### The Protocol Used to Verify the Authentication

Local Login - When a user performs a local login he usually also receives a ticket that will permit him to be authenticated to the services offered. The user, while logging in locally, is requesting a ticket-granting-ticket from the key-distribution-center (KDC) to use when requesting services from a particular server. This ticket-granting-ticket is presented to the ticket-granting-server (TGS) and the TGS issues a server-specific ticket. The server-specific ticket is then presented by the user for authentication to the server.

| | |
|---|---|
| 1.U $\rightarrow$ C: | $N_{client}$,password |
| 2. C $\rightarrow$ KDC: | c, tgs |
| 3. KDC: | 1. Generates session key $(SK)_{c,tgs}$. |
| | 2. Sets ticket start time, and expiration time for ticket. |
| | 3. Creates ticket($T_{c,tgs}$) as (s,c,addr,time,expir,$\{SK_{c,tgs}\}K_{tgs}$). |
| 4. KDC $\rightarrow$ C: | $\{SK_{c,tgs}\}K_c, \{T_{c,tgs}\}K_{tgs}$ |

<u>Where:</u>

| | |
|---|---|
| s | server |
| c | client |
| addr | source address |
| time | starttime |
| expir | lifetime |
| tgs | ticket granting server |
| $\{i\}K_q$ | "i" encrypted in a given key "q" |
| $T_{c,tgs}$ | ticket for "c" to use "tgs" |
| $SK_{c,tgs}$ | session key for "c" from ticket granting service |

In version 4, message 4 is: KDC $\rightarrow$ C: $\{SK_{c,tgs}, \{T_{c,tgs}\}K_{tgs}\}K_c$. The change results from a performance issue of the ticket being doubly encrypted (once in step 3 and again in step 4) when being sent from the KDC to the client. This forces the client to have to decrypt the ticket before presenting it to the server. There is no need to encrypt the ticket in the message from the KDC to the client, and doing so can be wasteful of processing time if encryption is computationally intensive (as will be the case for most software implementations. [Koh91, p4]. Not having to perform this "extra" encryption that results in more ciphertext alleviates some degradation of performance, without increasing risk significantly.

5. C: 1. Decrypts $SK_{c,tgs}$ using Kc. If this decryption fails,
   user is prompted for password again.
   2. Stores $SK_{c,tgs}$ and $\{T_{c,tgs}\}K_{tgs}$ for future use.
   3. Destroys password and $K_c$.

The client now possesses a session key for use between the client and the ticket-granting-service, along with a ticket-granting-ticket to present to the ticket-granting-service. For any further service requests, the client communicates with the ticket-granting-service, rather than the key-distribution-center. Also of note is that the user password is not needed for any further authentication. The ticket-granting-ticket in conjunction with the session key between the client and the ticket-granting-server are used for authenticating the client, who is acting on behalf of the user.

Remote Login and Client/Server Requests - These two types of accesses are treated the same. The user presents credentials and an authenticator to the ticket-granting-service requesting a ticket for a particular service. This may be for example, the telnet service, use of an "r-command" (rlogin, rsh, etc.).

| | |
|---|---|
| 1. C $\rightarrow$ TGS: | $\{A_c\}SK_{c,tgs}, \{T_{c,tgs}\}K_{tgs}$, s |
| 2. TGS : | 1. Verifies $T_{c,tgs}$, $A_c$, and accompanying server request. |
| | 2. Generates new ticket for use between client and requested server $(SK)_{c,s}$. |
| 3. TGS $\rightarrow$ C: | $\{(SK)_{c,s}\}SK_{c,tgs}, \{T_{c,s}\}Ks$ |
| 4. C: | 1. Decrypts $(SK)_{c,s}$ using $(SK)_{c,tgs}$ and stores session key and ticket for service requests to this particular server. |
| 5. C $\rightarrow$ S: | $\{A_c\}K_{c,s}, \{T_{c,s}\}K_s$ |

Where:

| | |
|---|---|
| $A_c$ | authenticator for "c" |
| s | server request |

The client makes a request to the ticket-granting-server by sending the ticket-granting-ticket, an authenticator encrypted under the session key shared between the client and the ticket-granting-server, and the server/service request. After verifying the authentication of the client, the ticket-granting-server sends to the client a session key to be shared between the client and requested server that will be used to encrypt the authenticator, and a ticket (encrypted under the server's secret key) that is required to be presented to the server. The client can now use these for the life of the ticket (or login session) to authenticate the user when server/service requests are made.

### The Principals

The principals involved in the Kerberos model are the user, the client, the key-distribution-center, the ticket-granting-service, and the server providing the requested service. The client acts on the user's behalf and allows the Kerberos communications and computations to be transparent to the user (unless, of course, there is an error, or a ticket expires). Both the client and the ticket-granting-service must trust that the key-distribution-center provided the client with the correct secret key of the user. Once the key-distribution-center provides the client with a ticket for the ticket-granting-service, the key-distribution-center need not be involved in further communications. The ticket-granting-service and the key-distribution-center usually reside on the same machine, with the ticket-granting-service having read-only access to the secret key database. This is so the ticket-granting-service can obtain a server's secret key in order to create a client/server ticket. Having these two principals residing on the same machine eliminates the need for the ticket-granting-service to obtain secret key information over the network, and takes advantage of the strong physical protection mechanisms used to protect the key-distribution-center.

### The Areas of the Network Where Trust is Placed

The key-distribution-center (KDC) stores all secret keys for all users and servers. This machine must be physically secured, as well as have strong access control mechanisms for updating the database of keys. Both clients and servers must trust that the information they receive from the key-distribution-center is correct. A major vulnerability with the Kerberos model is that if the key-distribution-server is compromised, every secret key used on the network is compromised.

### The Areas of the Network Where Secrets are Kept

The client stores the user's password and secret key for a period of time until the client receives a ticket-granting-ticket from the ticket-granting-service. After receiving this ticket, the client can destroy the copy of the password and secret key since they no longer need to be used. There is no copy of a password file on a client. This reduces some of the vulnerability of an intruder copying the password file and using a dictionary attack to obtain passwords. The tickets stored on a client are vulnerable to attacks on the client. Kerberos assumes protection of the tickets on the client by having only one user logged into a client at a time and by limiting the lifetime of the ticket. However, if a user can log in as *root*, he can then *su* as another user currently logged into the client and obtain his tickets for use until the lifetime of the ticket expires.

### The Key Generation and Distribution Model Used

The secret keys for users are generated based on a one-way function of the users' password. This is done so that the user is not required to remember a very long number (the secret key). However this creates a vulnerability for discovering the password. The user's secret key is used to encrypt the session key to be used between the user and the ticket-granting-service and clientname, servername, timestamps, etc. The vulnerability is created because this information, when decrypted, results in plaintext. Since the keyspace for a user's key can be considered smaller than all possible DES keys (because it is based on the user's password, a limited pool to choose from) an imposter could capture a response from the ticket-granting-service to the user, and perform a dictionary attack to generate the correct key. The correct key (and thus password) is found when the message decrypts into readable form. Kerberos Version 5 has made provisions for the use of smartcards or tokens that can be used to store a user's key, thus eliminating the need generating the user's key based on his password.

The key distribution model used in Kerberos is based on the Needham and Schroeder key distribution protocols, modified with the addition of timestamps [SMS87, p.7]. A tutorial paper by Voydock and Kent provides an introduction to the topic and explains the timestamp modifications.

### The Composition of the Ticket/Certificate

In Kerberos there are two items needed to prove authentication. The first is the ticket, the second is the authenticator. The ticket consists of the requested servername, the clientname, the address of the client, the time the ticket was issued, the lifetime of the ticket, the session key to be used between the client and the server, and some other fields. The ticket is encrypted using the server's secret key, and thus cannot be correctly decrypted by the user. If the server can properly decrypt the ticket, when it is presented by the client, and the client presents the authenticator encrypted using the session key contained in the ticket, the server can have confidence that the user is who he claims to be.

The authenticator contains the clientname, the address, current time, and some other fields. The authenticator is encrypted by the client using the session key shared with the server. The authenticator provides a time-validation for the credential. If a user possesses both the proper credential and the authenticator encrypted with the correct session key, and presents these items within the lifetime of the ticket, then the user's identity can be authenticated.

## 10.4.6 Secure RPC

Sun Microsystems calls the authentication mechanism, used in their Secure RPC service, DES Authentication. "The security of DES authentication is based on the sender's ability to encrypt the current time, which the receiver can then decrypt and check against its own clock. The timestamp is encrypted with DES. Two things are necessary for this scheme to work:

1. the two agents must agree on what the current time is, and

2. the sender and receiver must be using the same encryption key."[SUN90b, p.429]

### The Protocol Used to Verify the Authentication

Local Login - When a user presents his userid and password to a client, the client may proceed with the local login mechanism. That is, the client may have a local password file and compare a one-way function of the user supplied password to the encrypted password stored locally. Alternatively, the client may not store passwords locally and may need to request the one-way encryption of the user's password from a centralized database that contains usernames and one-way encryptions of the passwords. This is usually the same database that contains the network public-key database. Along with authenticating the user locally, the client also receives from the public-key database the public and private keys for the user. The private key is encrypted using the user's password. (The vulnerability created by this is discussed below.) The client decrypts the private key using the user-supplied password. The client stores the secret key (probably encrypted under a client secret key, although this was not verified in the literature) for future server requests. It should be noted that the user's password is never sent across the network.

| | |
|---|---|
| U → C: | U, PW |
| C: | 1. Retrieves from public key database a user record containing: username, user's public key, {user's secret key}$_{PW}$<br>2. Decrypts secret key using PW and stores secret key in the keyserver process |

Where:

| | |
|---|---|
| U | user |
| PW | password |

Remote Login and Client/Server Requests - The protocol used to authenticate users who request these two types of accesses is the same. The protocol used to authenticate the client, acting on behalf of the user, is shown below.

| | |
|---|---|
| 1. C: | 1. Receives server's public key from public-key database.<br>2. Generates session key $(SK)_{c,s}$ for use between client and server. |
| 2. C → S: | c, $\{CK\}_{SK(c,s)}, \{window\}_{CK}, \{t_1, window + 1\}_{CK}$ |
| 3. S: | 1. Receives client's public key from public-key database.<br>2. Generates session key $(SK)_{c,s}$ for use between client and server.<br>3. Decrypts the conversation key (CK) by using SK.<br>4. Decrypts the $t_1$, window and window+1.<br>5. Stores into a credential table, with an index (ID): C, CK, window, $t_1$. |
| 4. S → C: | $\{t_1 - 1\}CK, ID$ |
| 5. C: | Stores ID and CK in key server process. |

Where:

| | |
|---|---|
| c | client |
| CK | conversation key |
| window | lifetime of CK |
| $SK_{c,s}$ | secret key generated by client and server |
| $t_1$ | original timestamp |
| $t_n$ | current timestamp |

For any further requests between the client and server:

| | |
|---|---|
| 6. C → S: | $ID, \{t_n\}_{CK}$ |
| 7. S → C: | $\{t_n - 1\}_{CK}, ID$ |

1. & 2. The client, acting on the user's behalf, authenticates the user to the server. To accomplish this, the client:

1. creates a conversation key (CK) to be kept secret between the client/user and the server. CK is DES encrypted using the secret key (SK) independently generated by the client and server (an explanation of how this key is generated is below).

2. defines a window of time that specifies how long CK can be used.

3. provides a current timestamp.

4. specifies window+1.

The timestamp, window and window+1 are all encrypted under the conversation key.

3. The server decrypts the conversation key (CK) by using the secret key (SK). The server can then decrypt the timestamp, the window and the window+1 using the conversation key. These values are stored by the server into a credential table. The window+1 value helps to prevent a correct chance guessing by an intruder a correct timestamp/window combination. (Details on this type of attack were not provided).

The file server knows that the user is who he claims to be, because according to [GS91, p. 284]:

- The packet that the user sent was encrypted using a conversation key.
- The only way that the user could know the conversation key would be by generating it, using the server's public key and the user's secret key.
- To know the user's secret key, the workstation had to look up the secret key in the public key database and decrypt it correctly.
- To decrypt the encrypted secret key, the user had to have known the key that it was encrypted with - which is, in fact, the user's password.

4. The server sends the index (ID) back to the client, along with the timestamp-1 encrypted. This timestamp value verifies the identity of the server to the client.

5. The client stores ID and CK to use for future requests to the server. The ID/CK pair stored on the client is deleted when the user logs out.

6 & 7. For future requests, the client authenticates the user to the server by sending the ID, and a current timestamp encrypted using CK. The server decrypts the timestamp, uses the ID to find the entry in the credential table, and determines if the current timestamp is not beyond the original timestamp+window. If the timestamp is valid, the server verifies the authentication of the user, and sends back to the client the timestamp-1, encrypted using CK. The client knows this response came from the server, because the server had the correct value for timestamp-1, and encrypted this value with the correct key.

### The Principals

The principals involved in the Secure RPC authentication system are users, clients, servers, and the authentication server. Secure RPC should be transparent to the user. The user supplies his password once and is authenticated for server usage based on the original password. The client acts on behalf of the user for the protocol exchanges. Servers provide the requested services and require that users be authenticated before providing the services. The authentication server provides the public and private keys to servers and clients.

### The Areas of the Network Where Trust is Placed

There is a server on the network that contains all the public and private keys for all users and servers. This is called the public-key database and usually resides on the same machine as the network name-server. The private keys stored on this server are encrypted. The users' private keys are encrypted under the users' passwords. The server private keys are probably not encrypted under a password (although documentation on this was not found). Clients must trust that the private/public key pair given to them for a user is valid. Servers must trust that the user's public key that they obtain is also valid.

### The Areas of the Network Where Secrets are Kept

The authentication server that contains the private/public key pairs must be protected. Compromise of this server could compromise the security of all keys. During a given session, the client holds the conversation key (CK) and index into the server for each server request the user makes. The client also stores the user's secret key in order to make new server requests. There is an assumed given with this scenario that only one user is using the client at a time, and that this user can also become *root* on the client. If a user is using someone else's client machine, then that *root* user can *su* to the other user and utilize those tickets. This is not necessarily a protocol problem, but more a problem inherent in the client protection mechanisms.

During a given user session, the server also contains secret information. The server must contain the conversation key to use with the client. This key is kept in the keyserver process and is subject to normal operating system protections.

### Key Generation and Distribution Model Used

Secure RPC uses the Diffie-Hellman key generation method. Under this method, each user has a private/public key pair. A secret key, to be shared between the two users, is generated independently by each user. The key is generated by each user applying his own private key (known only to the owner) and the other user's public key. Fifty-six bits of this key are extracted and used as a DES key.

To perform a dictionary attack to decrypt the private key which is based on a user's password, the spoofer would have to send a request to a server to generate the session key and compare what the intruder generated with what the server generated. Hopefully, robust logging and monitoring procedures would not permit multiple failures from the many tries that this type of attack would produce.

### The Composition of the Ticket/Certificate

Secure RPC tickets, once the initial authentication is established from the client, contains the index (ID) into the server's credential database, and a timestamp encrypted under the conversation key (CK). If an ID and CK can be determined, an imposter can pose as the legitimate user until the usage time expires, based on the window.

### 10.4.7 Concerns with Kerberos and Secure RPC

**Secure RPC**

- A dictionary attack can be performed on the user's private key. However, an attacker can only verify that he has guessed the correct password, and hence the correct key, by requesting a service. If a service refuses additional requests after multiple failures, or if the service is audited sufficiently, this can deter the risk of this type of attack succeeding.

- The user's secret key is kept in the memory of the keyserver from login to a key logout. If privileges are not controlled properly, meaning that another user can become superuser and then in turn become another user, the system has no way of knowing that an intruder is using the secret key of another user.

- Conversation keys are kept in processes on both the client and server. These keys then must rely on the protection of both systems. If one system stringently protects the key, but the other system does not, the protection efforts of the first system are reduced.

**Kerberos**

- A dictionary attack on the response from a Kerberos server is possible since the ticket-granting-ticket is returned encrypted using the user's password as a key. The response to the user contains readable information. An attacker may capture this response and perform a dictionary attack on it until readable plaintext is produced.

- Tickets are kept in the memory on both clients and servers. The protection of these tickets is then left to the strength of the protection of the systems.

- Kerberos can be susceptible to a single-point-of-failure attack since both clients and servers must rely on a Kerberos server to be granted and to verify tickets. The Kerberos server's ability to authenticate users with trust essentially relies on one master key.

## 10.5 Using Robust Authentication Methods

Despite some weaknesses, the Kerberos and Secure RPC authentication mechanisms described in the previous section do much to prevent impersonation in a network environment. Note that Kerberos and Secure RPC only provide authentication services. Confidentiality and integrity services must be provided by other means. In this section, techniques for using Kerberos and Secure RPC to improve the authentication used by common network services are described. The description of these techniques is presented by means of examples of systems that are currently available.

### 10.5.1 Example Scenario

It is currently possible to set up a network environment in which all of the major network services, except for electronic mail, are authenticated using the Kerberos or Secure RPC mechanisms. These network services include: remote login, remote execution, file transfer, and transparent file access (i.e., access of remote files on the network as though they were local).

Authentication is important with electronic mail. When mail is sent, it is important that no one can send mail under a name other than the actual author of message sent. However, this scenario does not address the problems of electronic mail. For a discussion of electronic mail, see chapter 11 and sections 9.2.4 and 10.2.4.

This scenario assumes that the network environment consists of two kinds of systems: client workstations and server systems. Client workstations access network services from the servers. Servers are administered by responsible individuals whose job is to provide network services to the client workstations. The user of the client workstation is authenticated by means of Kerberos or Secure RPC for every service accessed on a server. No use of a server is permitted without that use being authenticated by Kerberos or Secure RPC.

As noted in section 10.4.7, with both the use of Kerberos and Secure RPC, there is the possibility of a workstation owner and/or administrator using *su* to impersonate another user who may be logged into the client workstation. Client workstations may be used either by a single individual or by several individuals. When a workstation is used by a single individual, that individual is typically the owner/administrator of the workstation and, in this scenario, no other user is permitted access. When a client workstation is used by several individuals, then the workstation is administered by a responsible administrator and, in this scenario, the workstation is configured so that no user may perform an *su*. This reduces the possibility of the workstation owner and/or administrator using *su* to impersonate another user who may be logged into the owner's workstation.

This scenario greatly reduces the threat of impersonation over a network as compared to the traditional practices such as password only authentication and trusted hosts. With Kerberos and Secure RPC, passwords are not transmitted over the network in plain text. Discovering passwords by intercepting packets is not easily accomplished but remains a potential threat. As described in the previous section, the potential threat of discovering passwords by intercepting packets is greater with Kerberos than with Secure RPC. However, both Kerberos and Secure RPC use techniques which make impersonation by means of packet replay or packet modification very difficult if not virtually impossible.

### 10.5.2 Scenario Implementation

The SunOS 4.x and Solaris 2.x operating systems from Sun Microsystems and the Kerberos Version 5 distribution from MIT are examples of systems which may be used to implement the scenario described above. Both systems from Sun Microsystems provide Secure RPC. In addition, Solaris 2.x provides Kerberos. The MIT Kerberos Version 5 distribution contains network service applications which use Kerberos for authentication and tools to develop such

Table 10.1: Network Services provided with some currently available systems

| Service | SunOS 4.x Secure RPC | Solaris 2.x Secure RPC | Solaris 2.x Kerberos | MIT Kerberos |
|---|---|---|---|---|
| Remote Login | *on -i/rpc.rexd* or *on/rpc.rexd* with *xterm* | *on -i/rpc.rexd* or *on/rpc.rexd* with *xterm* | *rlogin* | *telnet* |
| Remote Execution | *on/rpc.rexd* | *on/rpc.rexd* | *rsh* | |
| File Transfer | *on/rpc.rexd* with *cp* and NFS | *on/rpc.rexd* with *cp* and NFS | *rcp* | |
| Transparent File Access | NFS | NFS | NFS | |

applications on Sun Systems and other Unix based systems. Remote login, remote execution, file transfer, and transparent file access can be provided with either Kerberos or Secure RPC in Solaris 2.x. These services, authenticated with Secure RPC, are provided in SunOS 4.x. How each of these services authenticated with Kerberos or Secure RPC are made available in each of the systems is summarized in table 10.1.

### SunOS 4.x Secure RPC

SunOS 4.x already has a very large user community. The system provides a transparent file access service by means of a Secure RPC implementation of client and server NFS. The use of Secure RPC for NFS is a great improvement over the user ID/group ID authentication mechanism, referred to as "UNIX authentication," used in most NFS configurations (see sec. 9.2.8 and sec. 10.2.8).

A remote execution service, similar to *rsh*, is provided by *rpc.rexd*. The server process *rpc.rexd* must be configured in the inetd.conf file with the "-s" option to insure that Secure RPC is used for authentication. Without the "-s" option, *rpc.rexd* is invoked using only the user ID/group ID as a means of identifying the user. As is the case with NFS, using the user ID/group ID as a means of authentication is a weak mechanism. When used with *rpc.rexd*, the vulnerability is even greater. With NFS, the ability to authorize client systems to mount exported file systems makes it difficult for unauthorized client systems to exploit the weaknesses of the user ID/group ID authentication mechanism. When *rpc.rexd* is invoked without the "-s" option, any system on the network can attempt to exploit the weaknesses of the user ID/group ID authentication mechanism.

Some measure of protection can be achieved by using a "wrapper" (see sec. 10.3.2 and [LeF92]). The technique described in [LeF92] provides a capability for a *rpc.rexd* server to restrict access to specified clients. Note that the technique of [Ven92] is not applicable.

The technique of [LeF92] only restricts access by IP address. Once access is provided to a client workstation, the user ID/group ID is still used as an authentication mechanism. Thus, a client with access can still exploit the weaknesses of this authentication mechanism. With NFS, the server can be configured to export to a client only those files which belong to the owner of that client. Consequently, if the owner of that client should try to impersonate another user, no harm is done since only the owner's files are exported to the client. However, the *rpc.rexd* server (in the absence of the "-s" option) initiates commands on behalf of the user ID/group ID which initiated the call. The server implicitly trusts the user ID/group ID it is given.

In addition, the technique of [LeF92] does not protect against IP address impersonation. This is easy to accomplish on a subnet when a system is turned off frequently or its network connection is not functioning.

To summarize, *rpc.rexd* should almost never be run without some added security. The technique of [LeF92] offers some measure of protection by allowing only certain client systems access but provides no method of user authentication other than the default user ID/group ID mechanism. Using Secure RPC (*rpc.rexd* with the "-s" option) is the preferred method of providing proper user authentication.

The client side of *rpc.rexd* is the *on* command. The *rpc.rexd* server may also be called from an application program using the *rex* protocol defined in */usr/include/rpcsvc/rex.x*. When the *on* command is invoked, the current default directory on the *on* client must be part of a file system exported to the *rpc.rexd* server. Since this may not be desirable, implementing a version of *on* that does not require this may be required. Such an implementation is relatively straightforward [SUN90a]. An example of such an implementation is given in Appendix C.

A remote login service, similar to *rlogin*, can be provided in SunOS 4.x either by using the command:

*on -i <server>*

or by combining the capabilities of *on/rpc.rexd* and the X application *xterm*. The command:

*on <server> xterm <options>*

provides an *xterm* window to *<server>*. If *rpc.rexd* is configured with the "-s" option, then the user is authenticated to the server by means of Secure RPC. Using this method of providing a remote login capability implies that the X security mechanism used by the client workstation's X Server is SUN-DES-1 which uses the same facilities which support Secure RPC (see Chapter 7). While this is not required, using the SUN-DES-1 X security option with the X server is likely the most convenient. If some other X security option is configured, that choice should at least provide the robustness of the SUN-DES-1 option. Note that the SUN-DES-1 option requires the X11R5 X server. Neither Open Windows versions 2.x or 3.x provide an X11R5 X server. An X11R5 X server can be obtained from MIT.

While NFS implemented using Secure RPC should provide most of the requirements for a file transfer service, i.e., most files needed from a server could be at least exported read-only, some files located on a server may not be exported. A file transfer service can be provided by combining the capabilities of *on/rpc.rexd* and NFS. By using the remote login or remote execution services provided by *on/rpc.rexd*, a needed file can be copied to the user's exported file system. From there, the file can be copied to a local file system.

Although SunOS 4.x provides some services authenticated by Secure RPC, the NIS Name Service which is needed by Secure RPC is not provided with that level of authentication. This can be a serious weakness.

### Solaris 2.x Secure RPC

Solaris 2.x is the next major operating system release to follow SunOS 4.x. It is expected to be as widely used in the future as SunOS 4.x is now. Solaris 2.x provides all of the Secure RPC authentication services of SunOS 4.x. In addition, Solaris 2.x provides significant enhancements to the NIS Name Service which is called NIS+ in Solaris 2.x. Like NIS, NIS+ is built upon Sun's RPC but, unlike NIS, NIS+ uses Secure RPC for name service authentication.

### Solaris 2.x Kerberos

In addition to providing Secure RPC, Solaris 2.x also provides Kerberos. Remote login service is provided by a Kerberos authenticated *rlogin*. Remote execution service is provided by a Kerberos authenticated *rsh*. File transfer service is provided by a Kerberos authenticated *rcp*. Transparent file access service is provided by a Kerberos authenticated NFS.

### Kerberos from MIT

Source for a Kerberos implementation is available from MIT. The distribution includes a Kerberos server, applications which use Kerberos authentication, and libraries for use in developing applications which use Kerberos authentication. The current distribution, Version 5, also requires ISODE (the ISO Development Environment) in order to provide support for ASN.1 which is now used in the Kerberos protocol. The distribution may be installed on systems which do not provide any robust authentication mechanisms or to develop additional network services on systems which already have some robust authentication mechanism.

Two types of applications are included in the distribution: complete applications such as *telnet* and a POP (Post Office Protocol) server, and sample applications which may be used as templates to develop other Kerberos authenticated applications. The *telnet* application includes an option to encrypt all packets between the login client and server to insure confidentiality.

## 10.6 Network Security and POSIX.6/POSIX.8

The ISO/IEC 9945-1:1990 Standard "Portable Operating System Interface for Computer Environments (POSIX)," (referred to here as POSIX.1 [ISO90a]), defines a standard operating system interface and environment to support application portability at the source code level. It is intended to be used by both application developers and system implementors. [ISO90a, p.21].

POSIX.1 does not address networking issues. However, interfaces that allow for a distributed environment, i.e., mounted file systems, are not precluded. Security is somewhat addressed in POSIX.1. POSIX.1 supports security mechanisms similar to the mechanisms which are implemented on most Unix systems.

Networking and security are essential in the modern computing environment. To meet these needs, two additional POSIX working groups, POSIX.8 "Transparent File Access," [POS93] and POSIX.6 "Security Extensions" [POS92b] were created to develop and standardize interfaces to allow for a networked POSIX environment that utilized more robust security mechanisms. This section will briefly discuss these two emerging standards and present some issues that arise when both exist in the same environment.

### 10.6.1 POSIX.8 - Transparent File Access

The purpose of this amendment to POSIX.1 is to extend and circumscribe the file access aspects of the operating system interface to support file access mechanisms which are incapable of supporting the full behavior required by POSIX.1 or which provide access to files via a networked mechanism.

Transparent File Access (TFA) [POS93] is a specification of system services including file behavioral characteristics. With regard to file behavioral characteristics, the TFA Specification describes the behavior that an application can expect when manipulating a file.

The goal of the TFA Working Group is to provide a file use specification that:

1. Permits the use of the widest possible kinds of file systems which can resemble the file system of POSIX.1 so that most implementors and programmers can make use of the specification.

2. Allows an application to determine the behavior which it can expect when manipulating a file. In particular, the specification allows an application to determine when behavior is not in accordance with POSIX.1.

3. Provides the means for an application to simultaneously manipulate files whose access characteristics may differ since they reside in different file systems.

To be able to access systems that do not meet the specifications of POSIX.1, file characteristics of P1003.1 have been grouped into subsets thats allow non-POSIX.1 files to be described. The interfaces that interact with the file characteristics have been modified to accept these subsets of P1003.1 file characteristics. A system compliant with the POSIX.8

specifications must use files that can be described as either Full TFA, Core TFA, or Subset TFA.

Full TFA requires the specifications of POSIX.1.

Core TFA is the minimum set of characteristics that must be provided so that a file is usable according to the TFA Specifications. Core TFA:

1. Must support regular files.
2. Need not support execute/search permission bits.
3. Need not support file owner or file group class.
4. May result in files being created or modified as a result of a failed open().

Subset TFA refers to a set of file behavioral characteristics that at least conforms to Core TFA but does not conform to Full TFA. For a more complete discussion of Transparent File Access, see [OB91].

### 10.6.2 P1003.6 - Security Extensions

As stated in the IEEE Draft Standard P1003.6.1[4] Enhancements to Protection, Audit and Control Interfaces to the Portable Operating System Interface Standard, "The goal of this standard is to specify an interface to security functions for a POSIX system in order to promote application portability. The security mechanisms supported by this standard were chosen for their generality - they satisfy most of the key functional requirements prevalent in modern trusted systems. The specific interfaces defined were selected because they were perceived to be generally useful to applications (trusted and untrusted). Two mechanisms - discretionary access control and appropriate privilege - are defined specifically to address areas in the P1003.1 standard that were deferred to this standard." The interfaces specified can support the implementation of the following:

1. Access Control Lists (ACL)

   Specifications for an ACL mechanism is provided because it was felt that the permission bit mechanism provided by P1003.1 as the discretionary access control mechanism was not robust enough to meet certain security requirements. The permission bit mechanism does not provide access granularity to a specific user, nor does it not allow for additional file permissions beyond read, write and execute.

   The introduction of ACLs into the POSIX set of interfaces was planned for during the development of the base P1003.1 standard. While the permission bit mechanism is required by the P1003.1 standard, it also allows for an "additional access control mechanism." As stated in IEEE Standard 1003.1-1990 "an additional access control mechanism shall only further restrict the access permissions defined by the file permission bits." The ACL mechanism defined by P1003.6 was designed to coexist with

[4] This specification is now known as P1003.1e.

the permission bit mechanism on order to support backward compatibility with older applications and allow the use of either or both mechanisms. The P1003.1 interfaces that were designated to be used with the permission bit mechanism will also work with the ACL mechanism.

2. Security Auditing

   The interfaces to support a security auditing mechanism were designed to promote portability for two types of applications with respect to auditing. The first type of application is an audit tool that reads the audit data and incorporates it into meaningful reports. The second type of application is one that would generate audit data based on its interaction with the system. This type of application may be trusted or untrusted. The interfaces specify how data can be written to and read from the area where audit data is stored.

3. Fine-grained Privilege

   The privilege mechanism used on Unix systems today is a two-state mechanism. As superuser (UID 0), the user has all privileges. If the user has a UID that is not 0, then the user has no privileges. POSIX.6 developed interfaces that can support a fine-grained privilege mechanism. Privileges can be controlled on a per process level. The specification also defines privileges (generally read-overrides and write-overrides) that must be supported by the implementation.

4. Mandatory Access Controls (MAC)

   Specifications for a mandatory access control mechanism is provided for environments that require a mandatory access control policy. With this type of policy, the system determines object (file) access based on clearances of users and classifications of files. This policy is used primarily in Department of Defense (DOD) environments. The specifications provide for a labeling mechanism to be used on a per file basis. The interfaces standardize on how the label of a file can be created, read, or modified. It should be noted that MAC does not address mounted file systems, a major area of interest in this discussion.

5. Information Labels (IL)

   Information labels appear to be much like MAC labels. However they are not. Information labels describe the information contained in a file, whereas a MAC label defines the classification of a file. Information labels do not play a role in access decisions, they merely provide indications concerning the type of information contained in a file.

For a more complete description of P1003.6, see Chapter 4.

### 10.6.3 Issues of Using P1003.6 and P1003.8 in the Same Environment

There has been an issue raised concerning the compatibility of POSIX.6 and POSIX.8. It has been alleged that POSIX.6 and POSIX.8 are "fundamentally incompatible." The controversy arises over the relationship that POSIX.6 and POSIX.8 each have to POSIX.1. Briefly stated, POSIX.6 provides extensions to POSIX.1 and POSIX.8 makes optional some required features of POSIX.1. One of the goals of POSIX.8 is to permit access to as many file servers as possible even if those file servers cannot support all of the file characteristics required in POSIX.1.

The model that will most likely be used in a network environment is the client/server model. This model allows one system, a client, to access files and services on another system, the server. The most common use of this model is for a client to logically mount a file space on the server as a local drive on the client.

The first assumption that must be made here (and one that is not addressed by P1003.6) is that the requesting process is authenticated to be the claimed identity. Authentication is not addressed in P1003.1 or P1003.6. In a standalone environment, authentication is not an application portability issue. However in a networking environment, the case could be made that authentication is an application portability issue. P1003.6 did not originally address networking issues because P1003.6 interfaces are extensions to P1003.1, which did not address networking issues. This assumption of authentication, from a POSIX view, implies that it is a decision of the remote system (in this case the server) to add the additional requirement of authentication if need be.

The next issue common to both POSIX.6 and POSIX.8 is file access control. In order for a user to access a file from local or remote storage, the user must be granted permission from the file access control mechanism. The mandatory access control interfaces of POSIX.6 do not apply here because they specifically do not address mounted file systems. However the discretionary access control interfaces that support access control lists (ACLS) do apply. In POSIX.6, a file must have a group ID (GID) (which is required by POSIX.1) and may optionally have an access control list. In POSIX.8, a file may optionally have a GID. At first glance, the example implies a "fundamental incompatibility." However, the important thing to note in the example is that in POSIX.6, the access control list is optional for each file on an individual basis, not for all files taken together. Similarly, in POSIX.8, the GID is optional for each file on an individual basis. A resolution of the alleged "fundamental incompatibility" is obvious and natural, i.e., if a file has an access control list, then it must have a GID, and if a file does not have an access control list, then it need not have a GID. This solution works only because the POSIX.8 specifications map the different file characteristics onto the POSIX.1 characteristics that the POSIX.6 interfaces expect.

There are other POSIX specifications that modify/enhance POSIX.1. The intent for each such specification is to merge these into one POSIX specification. This is not simply an editorial task. While the Working Groups are careful to avoid "fundamental incompatibilities" between their specifications, ambiguities will most certainly arise when the merge takes place.

In the meantime, is it possible for an implementation to be both POSIX.6 and POSIX.8 compliant? Since the "fundamental incompatibilities" illustrated in the example have an obvious resolution, an implementation can be made both POSIX.6 and POSIX.8 compliant. The key is to realize that those features of POSIX.6 that are extensions to POSIX.1 are optional on a per file basis and those features of POSIX.1 that are made optional in POSIX.8 are optional on a per file basis.

# Chapter 11

# X.400 Message Handling Services

**Paul Markovitz**

## 11.1 Introduction

In 1984, the CCITT (Consultative Committee on International Telegraphy and Telephony)[5] approved the first version of the X.400 series of Recommendations [CCI88a], [CCI88b], [CCI88c]. The Recommendations defined a general purpose, store-and-forward, messaging service. In 1988, the CCITT updated the Recommendations to include, among other features, security services that protect messages against modification and disclosure, and allow communicating parties to authenticate their identities.

This chapter provides tutorial information about the 1988 X.400 security services. Section 11.1 introduces the chapter. Section 11.2 discusses cryptography as a tool to protect data transmitted over insecure channels. Beginning with section 11.3, all material is specific to the MHS (Message Handling System). Section 11.3 overviews the MHS. Section 11.4 describes its primary vulnerabilities. Section 11.5 describes the means by which security-relevant information is conveyed in the MHS. Section 11.6 details the X.400 security services that counter the vulnerabilities described in section 11.4, and section 11.7 concludes the chapter by discussing limitations in the 1988 X.400 security architecture. Additional reading on computer security can be found in [AMPH87], and additional reading on X.400 security can be found in [Ank92] and [CM89].

## 11.2 Cryptography Overview

Data communications channels are often insecure, subjecting messages transmitted over the channels to passive and active threats. With a passive threat, an intruder intercepts messages

[5] This organization is now known as the Telecommunications Standards Sector (TSS) within the International Telecommunicaitons Union (ITU).

to view the data. This intrusion is also known as eavesdropping. With an active threat, the intruder modifies the intercepted messages. An effective tool for protecting messages against the active and passive threats inherent in data communications is cryptography.

Cryptography is the science of mapping readable text, called plaintext, into an unreadable format, called ciphertext, and vice versa. The mapping process is a sequence of mathematical computations. The computations affect the appearance of the data, without changing its meaning.

To protect a message, an originator transforms a plaintext message into ciphertext. This process is called encryption or encipherment. The ciphertext is transmitted over the data communications channel. If the message is intercepted, the intruder only has access to the unintelligible ciphertext. Upon receipt, the message recipient transforms the ciphertext into its original plaintext format. This process is called decryption or decipherment. The encryption and decryption concepts are illustrated in figure 11.1.

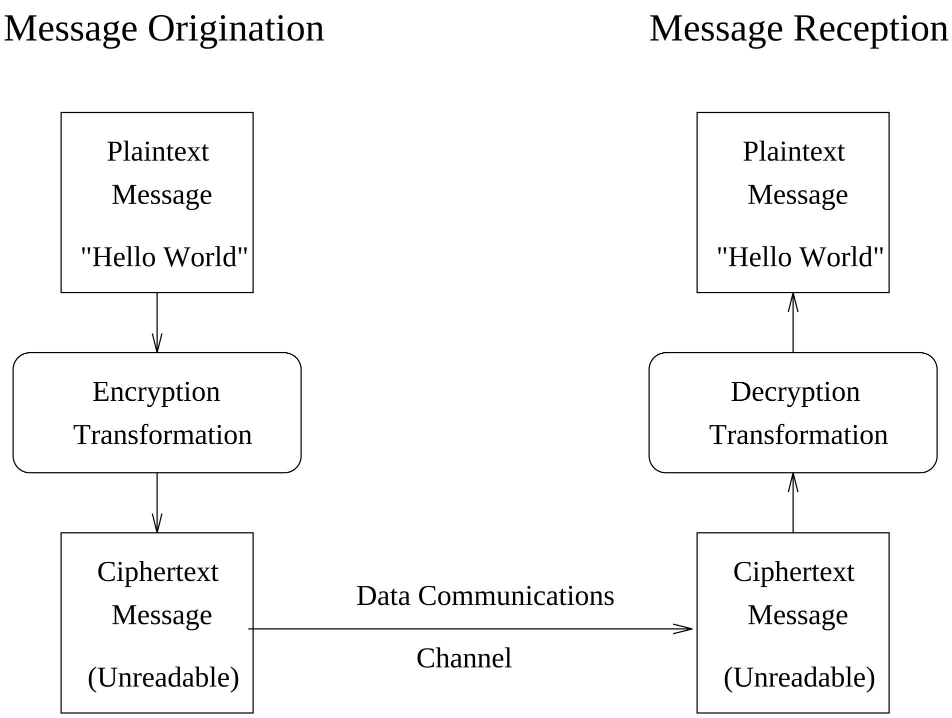



Figure 11.1: Message Encryption and Decryption.

The mathematical operations used to map between plaintext and ciphertext are identified by cryptographic algorithms. Cryptographic algorithms require the text to be mapped, and, at a minimum, require some value which controls the mapping process. This value is called a key. Given the same text and the same algorithm, different keys produce different mappings.

Cryptographic algorithms need not be kept secret. The success of cryptography is attributed to the difficulty of inverting an algorithm. In other words, the number of mappings from which plaintext can be transformed into ciphertext is so great, that it is impractical to find the correct mapping without the key. For example, the NIST DES (Data Encryption Standard) uses a 56-bit key. A user with the correct key can easily decrypt a message, whereas a user without the key would need to attempt random keys from a set of over 72 quadrillion possible values.

Cryptography is used to provide the following services: authentication, integrity, non-repudiation, and secrecy. Authentication allows the recipient of a message to validate its origin. It prevents an imposter from masquerading as the sender of the message. Integrity assures the recipient that the message was not modified en route. Note that the integrity service allows the recipient to detect message modification, but not to prevent it. There are two types of non-repudiation service. Non-repudiation with proof of origin provides the recipient assurance of the identity of the sender. Non-repudiation with proof of delivery provides the sender assurance of message delivery. Secrecy, also known as confidentiality, prevents disclosure of the message to unauthorized users.

Two approaches have been developed to provide the authentication, integrity, and secrecy services. Section 11.2.1 describes conventional or symmetric key cryptography. Section 11.2.2 describes public or asymmetric key cryptography. Section 11.2.3 discusses a scheme where the two cryptographic techniques are used together.

### 11.2.1 Symmetric Key Cryptography

Symmetric key cryptography is characterized by the use of a single key to perform both the encrypting and decrypting of data. Since the algorithms are public knowledge, security is determined by the level of protection afforded the key (i.e., ensuring that the key is known only to the parties involved in the communication). If kept secret, both the secrecy and authentication services are provided. Secrecy is provided, because if the message is intercepted, the intruder cannot transform the ciphertext into its plaintext format. Assuming that only two users know the key, authentication is provided because only a user with the key can generate ciphertext that a recipient can transform into meaningful plaintext.

The secrecy of the key does not ensure the integrity of the message. To provide this service, a cryptographic checksum, called a MAC (Message Authentication Code), is appended to the message. A MAC is a hashed representation of a message, and has the following characteristics:

- a MAC is much smaller (typically) than the message generating it,
- given a MAC, it is impractical to compute the message that generated it,
- given a MAC and the message that generated it, it is impractical to find another message generating the same MAC.

The MAC is computed by the message originator as a function of the message being transmitted and the secret key. Upon receipt, the MAC is computed in a similar fashion by the message recipient. If the MAC computed by the recipient matches the MAC appended to the message, the recipient is assured that the message was not modified.

Figure 11.2 illustrates the steps used to provide secrecy, authentication, and integrity in a conventional cryptosystem. It assumes the originator and recipient have agreed upon relevant algorithms and keys. In the figure the following conventions are used:

$M_{plain}$ a plaintext message,

$M_{cipher}$ the ciphertext message produced by encrypting $M_{plain}$,

**K** a secret key,

$\mathbf{E}(M_{plain},\mathbf{K})$ the encryption of $M_{plain}$ using K, and

$\mathbf{D}(M_{cipher},\mathbf{K})$ the decryption of $M_{cipher}$ using K.

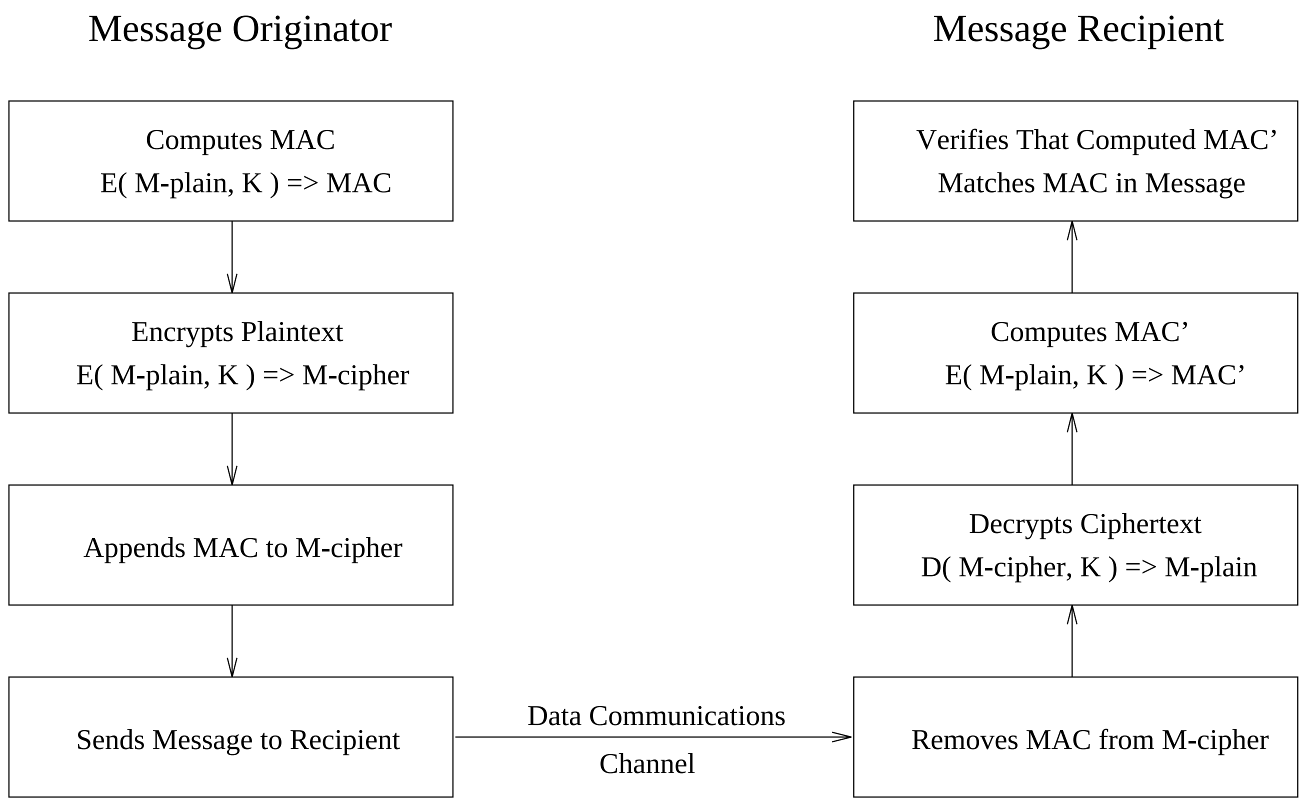



Figure 11.2: Security Services in a Conventional Cryptosystem.

Note that in figure 11.2, the message originator appended the MAC after encrypting the plaintext. If secrecy of the MAC is required, the MAC may be appended to the plaintext, and encrypted with it.

### Secret Key Distribution

The primary disadvantage of symmetric cryptography is the difficulty distributing the secret keys. A key cannot be transmitted securely over data channels, unless it is encrypted. Encrypting the key, however, requires another key. At some point, a plaintext key needs to be exchanged between communicating partners. One solution is to manually distribute the key (e.g., by registered mail). Manual distribution, however, is costly, time consuming, and prone to errors. Two automated approaches for distributing secret keys are discussed in this section: the ANSI (American National Standards Institute) standard X9.17, "Financial Institution Key Management" [ANS85] [FIP92], and the Diffie/Hellman key exchange.

ANSI X9.17 was developed to address the need of financial institutions to transmit securities and funds securely using an electronic medium. Specifically, it describes the means to assure the secrecy of keys.

The ANSI X9.17 approach is based on a hierarchy of keys. At the bottom of the hierarchy are data keys (DKs). Data keys are used to encrypt and decrypt messages. They are given short lifespans, such as one message or one connection. At the top of the hierarchy are key encrypting keys (KKMs). KKMs, which must be distributed manually, are afforded longer lifespans than data keys. Using the two tier model, the KKMs are used to encrypt the data keys. The data keys are then distributed electronically to encrypt and decrypt messages.

The two tier model may be enhanced by adding another layer to the hierarchy. In the three tier model, the KKMs are not used to encrypt data keys directly, but to encrypt other key encrypting keys (KKs). The KKs, which are exchanged electronically, are used to encrypt the data keys. Figure 11.3 illustrates the exchange of keys between two parties using the three tier model.

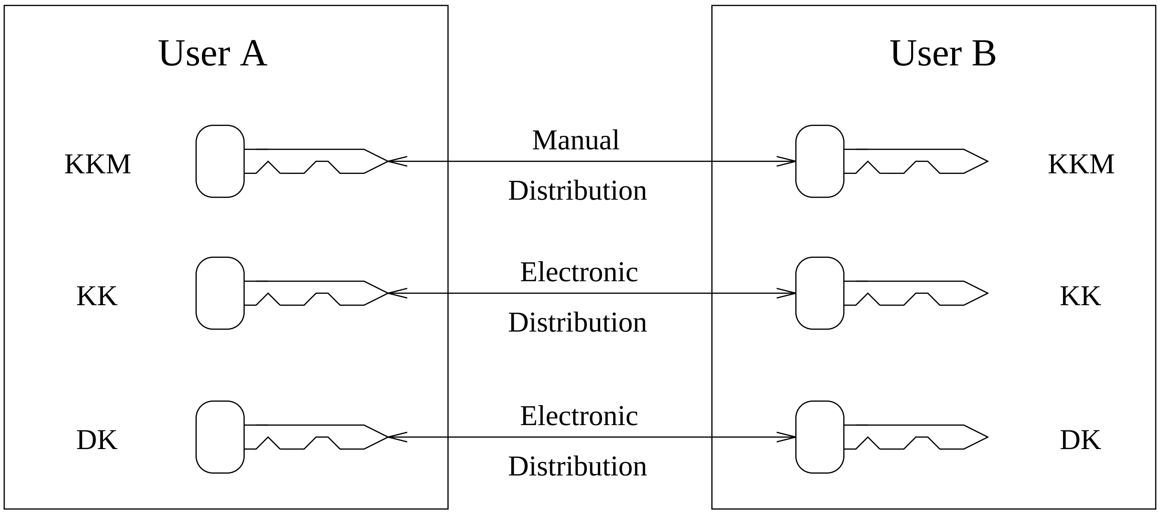



Figure 11.3: Point-to-Point Environment.

To exchange keys, one of the communicating parties creates a special message defined in X9.17, called a CSM (Cryptographic Service Message). CSMs are fixed-formatted messages used to establish new keys or discontinue use of existing keys. The CSM originator includes a MAC with the message (as specified in X9.9, "Message Authentication Standard" [ANS86]) to guarantee its integrity.

Figure 11.3 illustrates two users exchanging key material directly. This environment is known as *Point-to-Point*. The ANSI X9.17 standard describes two other environments for key distribution: *Key Distribution Centers* and *Key Translation Centers*. The key centers allow centralized management of keys. Rather than two parties sharing a KKM, each party shares a KKM with the center. The centralized management environment is shown in figure 11.4.

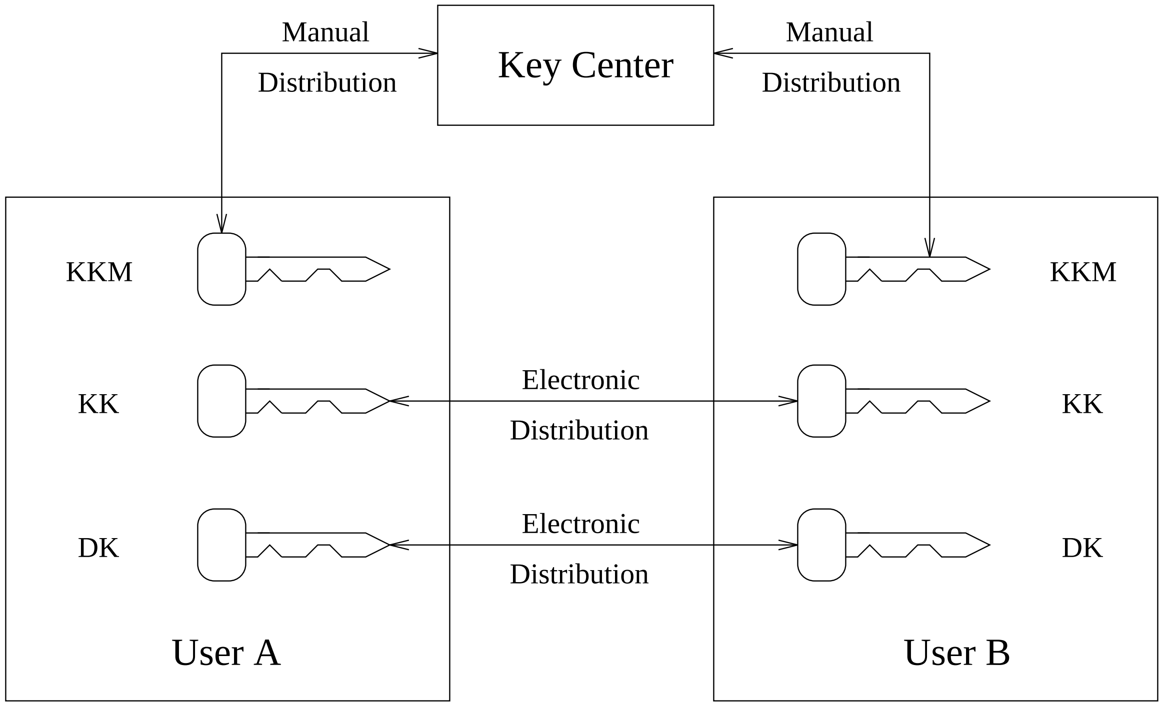



Figure 11.4: Centralized Management Environment.

The difference between the Key Distribution Center and the Key Translation Center is that the Key Distribution Center generates keys for its users. If an originator wants to send an encrypted message to a recipient, the originator submits the request to the Key Distribution Center. The Center generates and returns two identical keys to the originator. The first key is encrypted using the KKM shared between the Center and the originator. The originator decrypts the key, and uses it to encrypt the message. The second key is encrypted using the KKM shared between the Center and the recipient. The originator transfers this key electronically to the recipient. The recipient decrypts the key, and uses it to decrypt the originator's message.

Key Translation Centers are used when two parties require the key management functions provided by the center, but one or both of the parties want to generate the KKs and DKs. In this scenario, the originator submits a key and the recipient name to the Center. The

Center encrypts the key using the KKM shared between the Center and the recipient, and returns the encrypted key to the originator. The originator transfers the key electronically to the recipient.

The advantages of the key centers are flexibility and efficiency. Users only need to exchange and store one KKM (with the center), rather than one KKM per communications partner. The center administers the distribution of KKMs for all its users. One disadvantage of key centers is cost. Communication partners can reduce cost by first exchanging a KK with the aid of a key center, then distributing DKs using the *Point-to-Point* approach.

A different type of solution to the problem of secret key distribution is the Diffie/Hellman key exchange. The Diffie/Hellman key exchange allows certain information to be transmitted publicly, in order for two users to compute a shared key. The two users first agree upon a prime number and a primitive root, both of which may be public. Each user then selects a random number, computes some result based on the random number, the prime number, and the primitive root, and sends this result to the other user. Each user then performs one last computation based on the prime number, the user's own random number, and the result from the other user. This final computation yields a single value, which is the same for each user. This value can be used to generate secret keys.

The Diffie/Hellman key exchange is illustrated in figure 11.5. Boxes in the figure are divided into two parts: the top part describes the mathematical computation, and the bottom part applies the computation to example values. The example values are trivial; their purpose is to illustrate the technique. In an implementation, the prime number and primitive root would be of the magnitude $2^{512}$ to $2^{1024}$.

The security of the Diffie/Hellman exchange is based on the difficulty in computing discrete logarithms. In other words, knowing the public values (i.e., the prime number, p, and its primitive root, g), the value transmitted over the insecure channel (i.e., y), and that

$$\text{y} = g^x \text{ modulo p, for some x}$$

does not yield x, and thus, does not yield the key, K.

The Diffie/Hellman key exchange does not provide authentication. If in figure 11.5, an intruder intercepts y(B) and transmits a different value, *User A* would establish a secret key with the intruder, rather than with *User B*. The Diffie/Hellman procedure needs to be augmented with some authentication mechanism.

### 11.2.2 Asymmetric Key Cryptography

Asymmetric or public-key cryptography differs from conventional cryptography in that key material is bound to a single user. The key material is divided into two components:

- a private key, to which only the user has access, and
- a public key, which may be published or distributed on request.

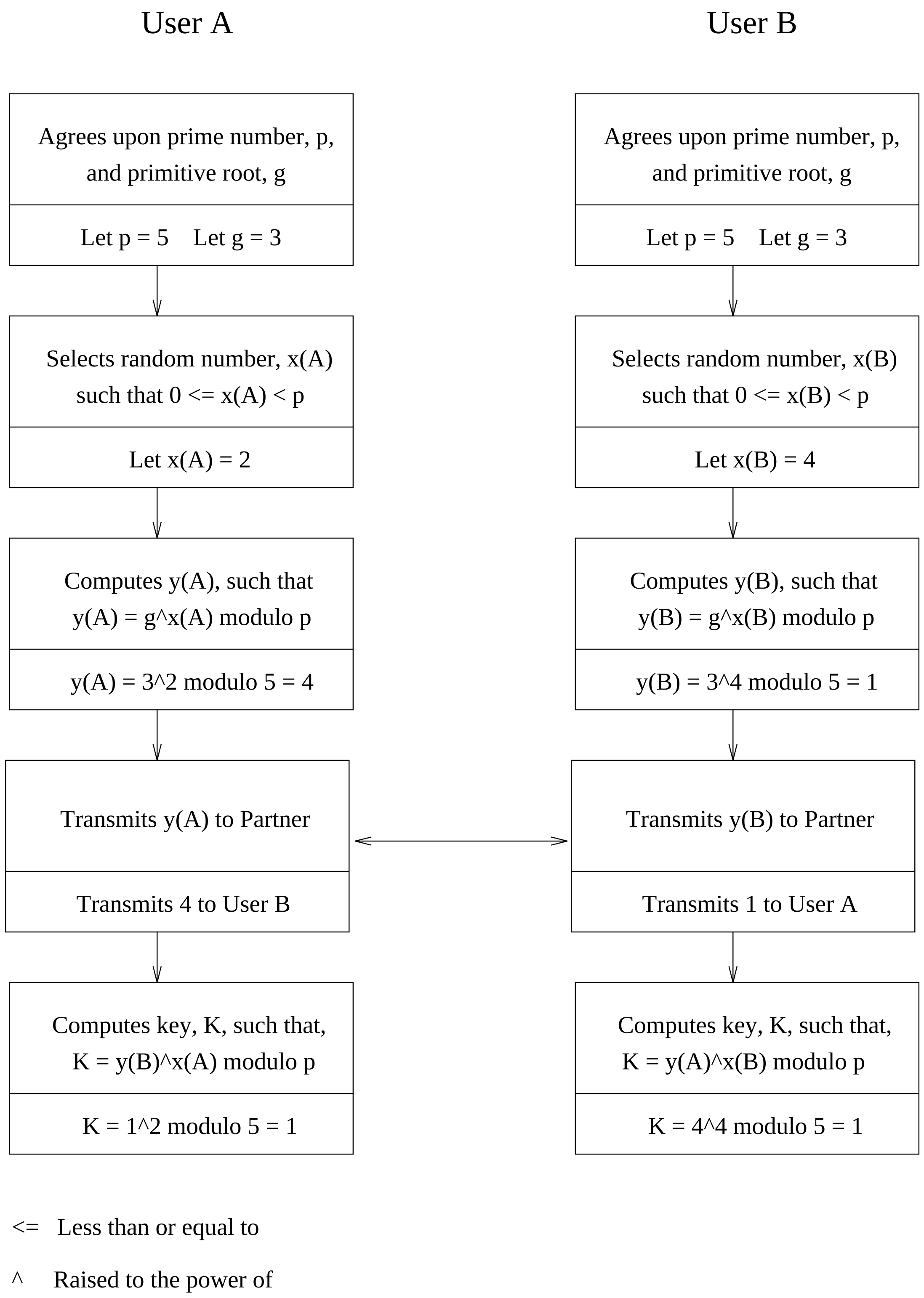



Figure 11.5: Diffie/Hellman Key Exchange.

Each key generates a function used to transform text. The private key generates a private transformation function, and the public key generates a public transformation function. The functions are inversely related, i.e., if one function is used to encrypt a message, the other is used to decrypt the message. The order in which the transformation functions are invoked is irrelevant. Note that since the key material is used to generate the transformation functions, the terms *private key* and *public key* not only reference the key values, but also the transformation functions. For example, the phrase, *"the message is encrypted using the message recipient's public key"*, means the recipient's public key transformation function is invoked using the recipient's public key value and the message as inputs, and a ciphertext representation of the message is generated as output.

The advantage of a public-key system is that two users can communicate securely without exchanging secret keys. For example, assume an originator needs to send a message to a recipient, and secrecy is required for the message. The originator encrypts the message using the recipient's public key. Only the recipient's private key can be used to decrypt the message. This is due to the computational infeasibility of inverting the public key transformation function. In other words, without the recipient's private key, it is computationally infeasible for the interceptor to transform the ciphertext into its original plaintext. Note that with a public-key system, while the secrecy of the public-key is not important (in fact, it is intended to be "public"), the integrity of the public-key and the ability to bind a public-key to its owner is crucial to its proper functioning.

One disadvantage of a public-key system is that it is inefficient compared to its conventional counterpart. The mathematical computations used to encrypt data require more time, and depending on the algorithm, the ciphertext may be much larger than the plaintext. Thus, the current use of public-key cryptography to encrypt large messages is impractical.

A second disadvantage of a public-key system is that an encrypted message can only be sent to a single recipient. Since a recipient's public key must be used to encrypt the message, sending to a list of recipient's is not feasible using a public-key approach.

Although public-key cryptography, by itself, is inefficient for providing message secrecy, it is well suited for providing authentication, integrity, and non-repudiation services. All these services are realized by the digital signature.

### Digital Signatures

A digital signature is a cryptographic checksum computed as a function of a message and a user's private key. A digital signature is different from a hand-written signature, in that hand-written signatures are constant, regardless of the document being signed. A user's digital signature varies with the data. For example, if a user signs five different messages, five different signatures are generated. Each signature, however, can be authenticated for the signing user.

Due to the efficiency drawbacks of public-key cryptography, a user often signs a condensed version of a message, called a message digest, rather than the message itself. Message digests are generated by hash functions.

A hash function is a keyless transformation function that, given a variable-sized message as input, produces a fixed-sized representation of the message as output (i.e., the message digest). For example, a hash function may condense a one-megabyte message into a 128 or 160-bit digest. For a hash function to be considered secure, it must meet two requirements; the hash function must be 1-way and collisionless. 1-way means that given a digest and the hash function, it is computationally infeasible to find the message that produced the digest. Collisionless means that it is not possible to find two messages that hash to the same digest. If a hash function meets the collisionless and 1-way requirements, signing a message digest provides the same security services as signing the message itself.

The following example describes the digital signature process. It assumes two users have agreed upon a hash function and a signature algorithm for the signature verification process. For clarity, message secrecy is not included in the example.

An originator needs to send a signed message to a recipient. The originator performs the following procedure:

- Generates a digest for the message.
- Computes a digital signature as a function of the digest and the originator's private key.
- Transmits the message and the signature to the recipient.

Upon receiving the message, the recipient performs the following procedure:

- Generates a digest for the received message.
- Uses this digest, the originator's public key, and the received signature as input to a signature verification process.

If the signature is verified, the following services are provided. First, the recipient is assured that the message was not modified. If even one bit of the original message was changed, the digest generated using the received message would cause the signature verification process to fail. Second, the recipient is assured that the originator sent the message. Public key transformation functions are 1-way (i.e., not forgeable); therefore, only a signature generated by the originator's private key can be validated using the originator's public key.

In addition to integrity and authentication, digital signatures provide non-repudiation with proof of origin. Non-repudiation with proof of origin is similar to authentication, but stronger in that the proof can be demonstrated to a third party. To provide authentication and non-repudiation with proof of origin using a digital signature, a message originator signs a message (or digest) using the private key bound to the originator. Since only the originator can access the private key, the signature is unforgeable evidence that the originator generated the message. In contrast, non-repudiation with proof of origin cannot inherently be provided in a conventional cryptosystem. Since both parties involved in a communication share a secret key, both parties can deny sending a message, claiming that the *other* party is the message originator.

In addition to the non-repudiation with proof of origin service, public-key cryptography has another advantage over conventional cryptography. The keys exchanged in a public-key system need not be kept secret. Thus, key distribution with a public-key system is simplified as compared to a private-key system.

### Public Key Distribution

Users of a public-key system must access the public keys of other users. One means to distribute public keys is certificates. A certificate is a public document containing information identifying a user, the user's public key, a time period during which the certificate is valid, and other information. Certificates are typically issued, managed, and signed by a central issuing authority called a CA (Certification Authority).

One method by which certificates can be distributed is described in the following example. *User A* and *User B* register with a CA. During the registration process, the users provide their public key information to the CA. The CA, in turn, provides each user with the following information:

- a signed certificate containing the user's public key, and
- the public key information of the CA.

The users store their certificates in a public directory (e.g., the X.500 Directory). At some future time, *User A* (the originator) sends a signed message to *User B* (the recipient). The message is signed using the originator's private key. Upon receipt, the recipient queries the public directory to obtain the originator's public key certificate. The recipient first uses the CA's public key to validate the certificate's signature, then verifies the originator's message signature using the public key contained in the certificate. One advantage of this scheme is that since public information is being transmitted, insecure data channels may be used for the communication. The digital signatures assure the integrity and authenticity of the information. This example is illustrated in figure 11.6.

In the above example, the two users were registered with the same CA. In practice, users may be certified by different CAs. In the case where two users who communicate frequently are certified by two different CAs, the CAs may certify each other. In other words, the two CAs may store each other's public keys in certificates signed by the certifying CA. This concept is called *cross certification*. In scenarios where there are large numbers of users and CAs, arranging the CAs in a hierarchy (see sec. 11.5.3) is more practical than requiring every CA to cross certify every other CA.

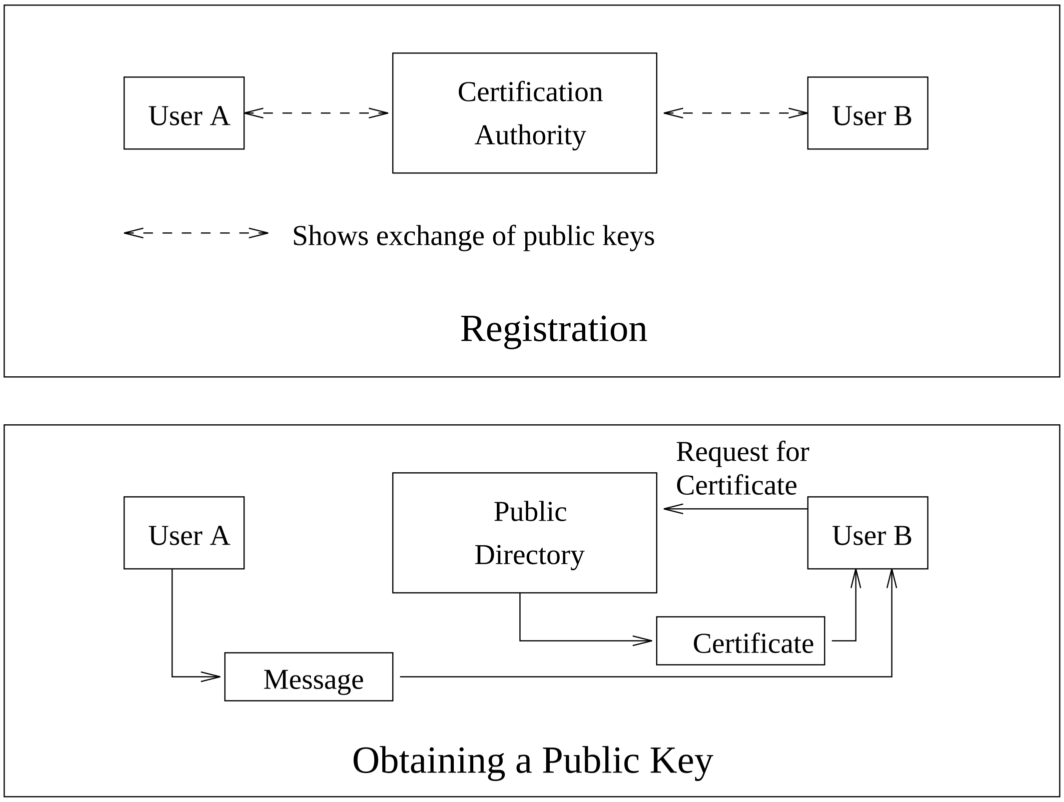



Figure 11.6: Key Distribution Using a Certification Authority.

### 11.2.3 Using Public-Key Cryptography for Secret Key Distribution

Public-key systems are inefficient for encrypting large messages. The secret keys used in conventional cryptography are characteristically small. If conventional secret keys are viewed as a kind of message, the encrypting of these keys using a public-key algorithm would not place an unnecessary burden on the processing of a computer system. Thus, the joint use of conventional and public-key cryptography can be used to provide authentication, integrity, and secrecy in an efficient manner. The following example illustrates this idea. Note that for simplicity, the example does not include the distribution of the public key certificates.

An originator needs to send a signed, confidential message to a recipient. The originator first computes a digital signature as a function of the originator's private key and a digest of the plaintext message. Second, the originator generates a conventional secret key, and uses this key to transform the plaintext into ciphertext. Third, the originator encrypts the secret key using the recipient's public key. The originator finally appends the encrypted secret key and the digital signature to the ciphertext, and transmits the information to the recipient.

Upon receipt, the secret key is decrypted using the recipient's private key. The secret key is then used to decrypt the ciphertext. Once the plaintext is obtained, the recipient validates the message signature as a function of the signature and the originator's public key. Secrecy is guaranteed, because only the recipient's private key can be used to decrypt the secret key needed to decrypt the message. Integrity is guaranteed because the digital signature was generated using a digest of the original plaintext message. Finally, authentication is achieved, because the digital signature provides unforgeable evidence that the plaintext message was generated by the originator. The step-by-step processing of this example is illustrated in figure 11.7.

This scheme addresses the two disadvantages of a public-key system: performance, and the inability to send a message to multiple recipients. Performance degradation is minimized, because a conventional algorithm (e.g., DES) is used to encrypt the message. Only the encrypting of the secret key (e.g., the DES key) requires a public-key algorithm. If the message is transmitted to several recipients, the originator encrypts the secret key one time per recipient, using that recipient's public-key. For example, if a message is sent to five recipients, five different encryptions of the secret key would be appended to the message.

## 11.3 X.400 Overview

This section provides an overview of the MHS (Message Handling System). Three aspects of the MHS are discussed: the function model, the message structure, and delivery reporting.

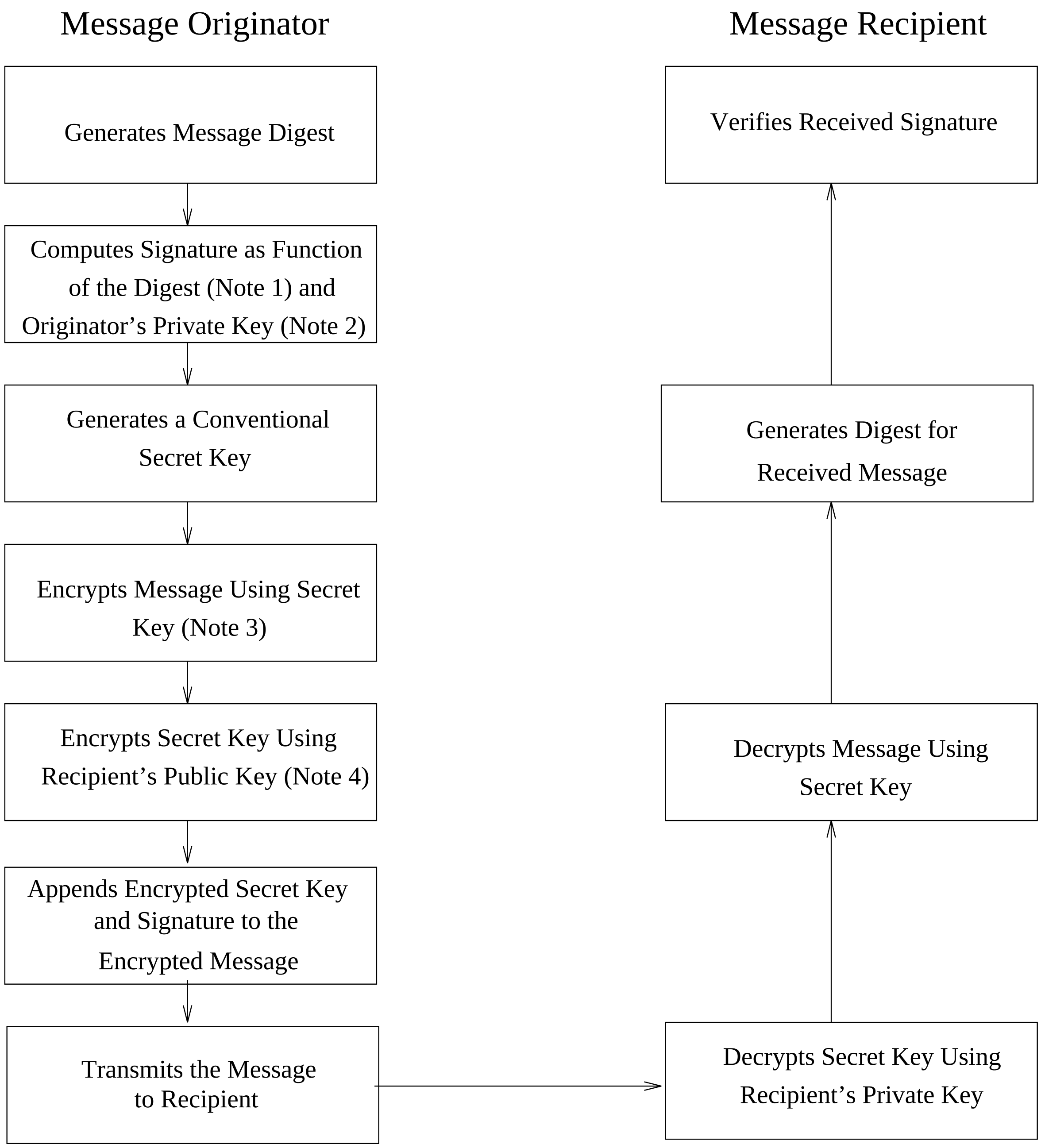
### 11.3.1 Functional Model

A functional model of the MHS is shown in figure 11.8. The MHS is a collection of MTAs (Message Transfer Agents), MSs (Message Stores), UAs (User Agents), and AUs (Access Units). MTAs perform the store-and-forward message transfer function. MSs provide storage for messages. UAs enable users to access the MHS, and AUs provide links to other communication systems (e.g., the postal system). A more detailed description of each of these entities follows.

MTAs comprise the MTS (Message Transfer System), the principal component of the MHS. A message is submitted to an MTA by an originating UA, MS or AU, transferred to the recipient MTA(s), and delivered to one or more recipient UAs, MSs, or AUs. If the message is addressed to multiple recipients, the appropriate MTAs perform any splitting (i.e., replicating) of the message needed for delivery to each recipient.

Messages are transferred between MTAs on a cooperating store-and-forward basis. Since no end-to-end association is required, the MTA serving the message recipient need not be active when the message leaves the originating MTA. The message may be stored at a relay (i.e., intermediate) MTA until the recipient MTA becomes operational.

MTAs transfer messages whose content may be encoded in any format. MTAs neither examine nor modify the content of messages except when performing conversion. Conversion increases the effectiveness of the MHS by allowing users to submit messages in one encoded format (e.g., telex), and have them delivered in another encoded format (e.g., IA5). A UA can register with the MTA the encoded information types that may be delivered, and request the MTA to perform any required conversions.




Note 1: Computing the signature using the digest provides the integrity service.

Note 2: Computing the signature using the originator's private key provides the authentication and non-repudiation of origin services.

Note 3: Encrypting the message provides the secrecy service.

Note 4: Encrypting the secret key using the recipient's public key guarantees that only the recipient can decrypt the key needed to decrypt the message.

Figure 11.7: Joint Use of Conventional and Public-key Cryptography.

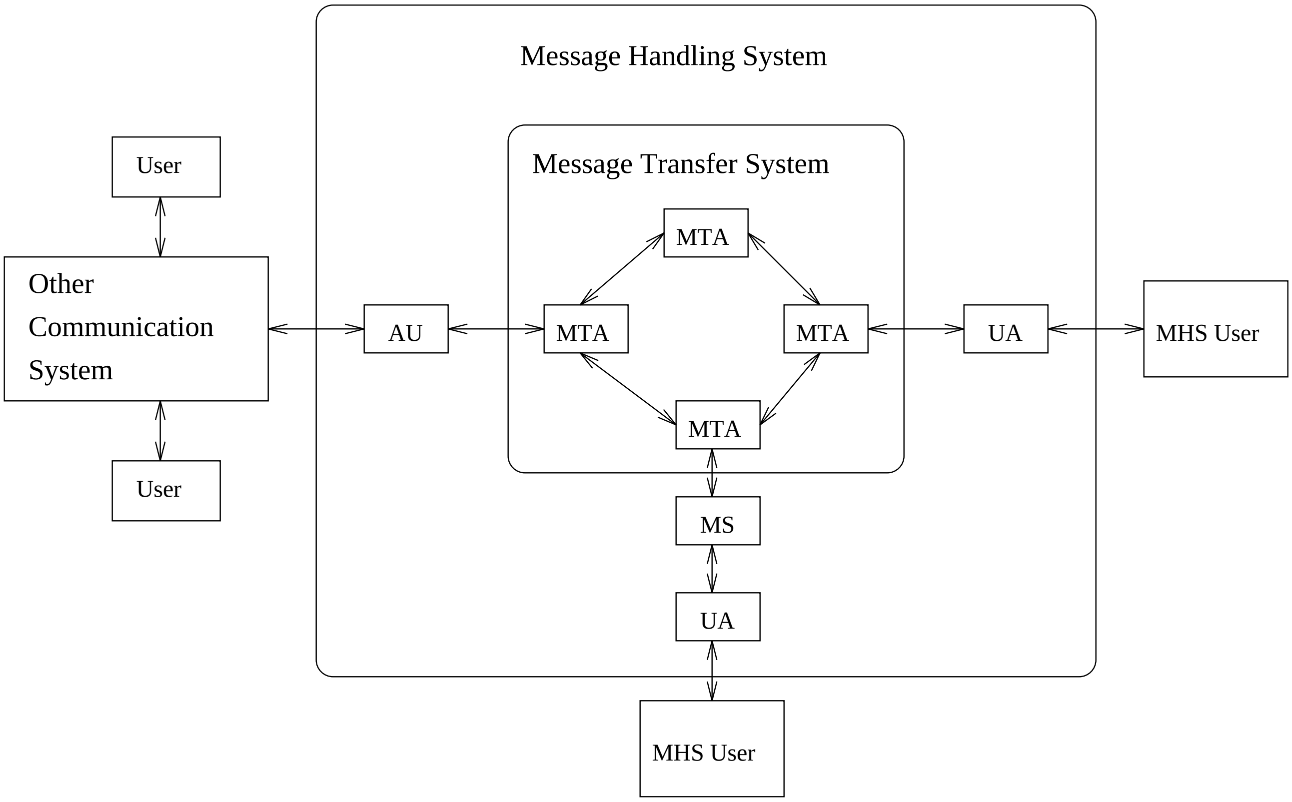



Figure 11.8: MHS Functional Model.

The UA is the MHS component that enables a user to access the MHS, for both the origination and reception of messages. When submitting messages, the UA supplies to an MTA, either directly or indirectly via an MS, the message content, the address(es) of the message recipient(s), and the MTS services being requested. The message content is the information that the originator wants transferred to the message recipient(s). The address(es) and service request data are used by the MTS to deliver the message. When receiving messages, the UA may accept delivery of messages directly from an MTA, or it may employ an MS to accept delivery of messages, and retrieve them from the MS at a later time.

The MS is an optional MHS component that acts as an intermediary between a UA and MTA. The MS often co-resides with the MTA serving it. The primary purpose of the MS is to provide a repository for the delivery of messages. The UA can retrieve messages from this repository. By using an MS to accept delivery of messages, a UA is not required to be constantly available. This is especially useful for UA applications implemented on personal computers, which are typically turned off at night. The MS may also submit and forward messages on behalf of the UA, and notify the UA at the time of message delivery.

The AU is the MHS component that provides a gateway between the MHS and another communications system. AUs may, for example, provide intercommunication with telex, teletex, and facsimile systems. Another AU, the PDAU (Physical Delivery Access Unit) enables MHS users to send messages to users residing on a physical delivery system, such as the Postal Service.

### 11.3.2 Message Structure

The structure of an MHS message is shown in figure 11.9. It consists of a message envelope and a message content. As with a postal message, the envelope represents the information required by the MTS to deliver the message, such as the address(es) of the recipient(s) and any special handling instructions. Many X.400 security parameters are transferred on the message envelope. The message content represents the information that the originator wants conveyed to the message recipient(s).

### 11.3.3 Delivery Reporting

The basic X.400 messaging service provides notification of message non-delivery. When a message cannot be delivered to a recipient, a non-delivery report is generated and returned to the originator. The content of the non-delivery report contains status information about the subject message.

The MT service also provides notification of delivery as an optional service. If a message originator requests acknowledgement of successful delivery, a delivery report is returned to the originator by the recipient's MTA upon delivery of the subject message.

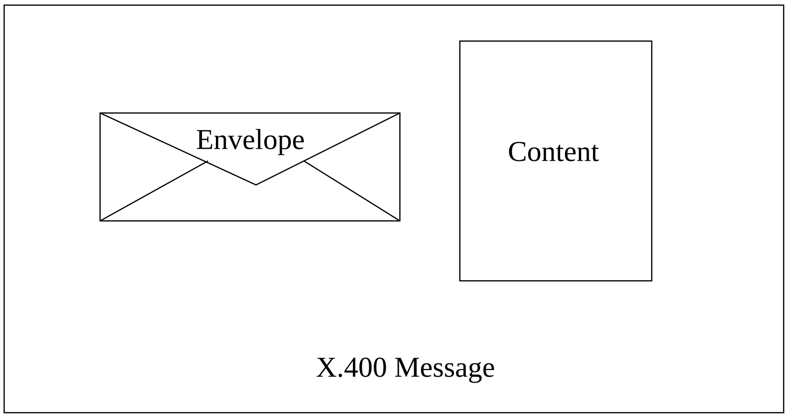



Figure 11.9: MHS Message Structure.

## 11.4 Vulnerabilities

The distributed nature of the MHS makes it vulnerable to various types of security threats. This section describes the nature of these threats, and concludes with a table that correlates each threat with the MHS security service that counters it.

The principal threats to the MHS can be divided into two categories, inter-message threats and intra-message threats. Inter-message threats arise from parties external to the message communication, and include: masquerade, message modification, message sequencing threats, and leakage of information.

Masquerade occurs when an entity successfully pretends to be a different entity. The following examples illustrate two types of masquerade to which the MHS is vulnerable. First, an unauthorized UA may impersonate an authorized UA to gain access to an MTA. Once access is gained, the unauthorized UA can falsely originate messages, falsely acknowledge receipt of messages, or simply discard messages. Second, an unauthorized MTA can impersonate an authorized MTA to misroute messages, or discard messages submitted for delivery.

Message modification occurs when a message is changed by an unauthorized party. Unauthorized changes apply to the message content, addressing information, security labels, and other message attributes. This threat also includes the destruction of an entire message.

Message sequencing threats jeopardize the ordering of messages. They include the reordering and replaying of messages.

Leakage of information occurs when an unauthorized party gains information by monitoring transmissions. The unauthorized party can learn of the content of messages, or of the parties involved in the message transfers. An unauthorized party can also gain information by analyzing the message traffic between two users.

The second category of threats is intra-message threats. Intra-message threats are those performed by the parties involved in the message communication. Intra-message threats include repudiation and security level violations.

Repudiation occurs when an MTA or MTS user (i.e., a User Agent, Message Store, or

Access Unit) denies performing a specific action. Repudiation threats include an MTS user denying the origination or delivery of a message, and an MTA denying the submission of a message.

Security level violations are threats relating to security labels. Security labels are data structures which permit the classification of a message, or a communicating party within the MHS, in terms of a security level (e.g., "Secret"). An example of a security level violation is an originator submitting a message with a security label that it is not authorized to generate.

The threats described above are reproduced below in table 11.1. Associated with each threat is the MHS security service or services that counter it. These services are described in detail in section 11.6.

Table 11.1: MHS Threats and Their Countermeasures

| | |
|---|---|
| *Masquerade* | |
| Impersonation and misuse of the MTS | Message origin authentication<br>Secure access management |
| Falsely acknowledge receipt | Proof of delivery |
| Falsely claim to originate a message | Message origin authentication |
| Impersonation of an MTA to an MTS user | Proof of submission<br>Report origin authentication<br>Secure access management |
| *Message Sequencing* | |
| Replay of messages | Message sequence integrity |
| Reordering of messages | Message sequence integrity |
| *Modification of information* | |
| Modification of messages | Content Integrity |
| Destruction of messages | Message sequence integrity |
| *Leakage of information* | |
| Loss of confidentiality | Content confidentiality |
| Loss of anonymity | Message flow confidentiality |
| Traffic analysis | Message flow confidentiality |
| *Repudiation* | |
| Denial of origin | Non-repudiation of origin |
| Denial of submission | Non-repudiation of submission |
| Denial of delivery | Non-repudiation of delivery |
| *Security level violations* | |
| Originator not cleared for security label | Secure access management<br>Message security labelling |
| MTA/MTS user not cleared for security context | Secure access management |
| Misrouting | Secure access management<br>Message security labelling |

## 11.5 Security-relevant Data Structures

Before describing how X.400 security services counter the threats presented in Section 11.4, three data structures must be discussed. These data structures: the security label, the asymmetric token, and the public key certificate, are used to convey security-related information between communicating parties. This section only defines the principal attributes comprising the structures; it provides no details regarding how these structures are used by X.400 security services.

### 11.5.1 Security Label

A security label is a collection of attributes associated with an MHS message or entity which permits its classification in terms of a security level. The security label attributes include:

**a security policy identifier** which identifies the security policy with which the security label is associated,

**a printable privacy mark** which identifies the level of privacy to be afforded a message or report (e.g., "In Confidence", "In Strictest Confidence"),

**a security classification** which classifies a message or report for security purposes (e.g., "Unclassified", "Confidential", "Top Secret"),

**a set of security categories** which restricts the context of the privacy mark, the security classification, or both. The categories are application-defined, and may include codewords or caveats to the privacy mark or security classification (e.g., "Personal-", "Staff-", "Commercial-").

Security labels may be transferred in MHS messages and reports, conveyed during the association establishment between two MHS entities (e.g., a UA may transfer security labels when connecting to its MTA), or registered with MHS entities (e.g., an MTA may maintain a registry of security labels for its users).

### 11.5.2 Asymmetric Token

The asymmetric token is a signed data structure used to convey security-related information from an originator to a recipient. The attributes comprising the token include:

- the name of the recipient,
- the date and time the token was generated,
- a collection of additional fields that is signed (signed-data):
  - a content confidentiality algorithm identifier,

  - a content integrity check,
  - a message security label,
  - a request for proof of delivery,
  - a message sequence number,
  - a non-repeating number,

- a collection of fields that is encrypted (encrypted-data):
  - a symmetric key used to encrypt the content,
  - a symmetric key used to compute a content integrity check,
  - a content integrity check,
  - a message security label,
  - a message sequence number.

The asymmetric token provides three forms of cryptographic protection. First, it ensures that only the recipient can view the plaintext information in the encrypted-data. This is because the token originator encrypts the encrypted-data using the recipient's public key. Thus, only the recipient's private key can be used to decrypt the information. Second, it ensures that the token has not been modified. Since the originator signs the token, the recipient can validate the signature and confirm the token's integrity. Third, it authenticates the identity of the token originator. This is because the originator signs the token using its private key. If the recipient validates the signature using the originator's public key certificate, only the originator's corresponding private key could have generated the signature.

X.400 defines two purposes for asymmetric tokens. They can be transferred as credentials when an MHS entity initiates a connection to a peer, and wants to provide strong authentication information. For this purpose the token is referenced as a *bind* token. Tokens can also be transferred in MHS messages, such that a distinct token can be generated for each message recipient. For this purpose the token is referenced as a *message* token.

### 11.5.3 Public Key Certificates

In order for a public key scheme to be successful, users must be guaranteed that the public key of another user truly belongs to that user. The means by which a public key is bound to a user is the public key certificate. The public key certificate is a collection of information issued and signed by a CA (Certification Authority). A certificate contains:

- the owner's public asymmetric encryption key,
- the DN (Distinguished Name) of the owner,
- the DN of the certification authority,

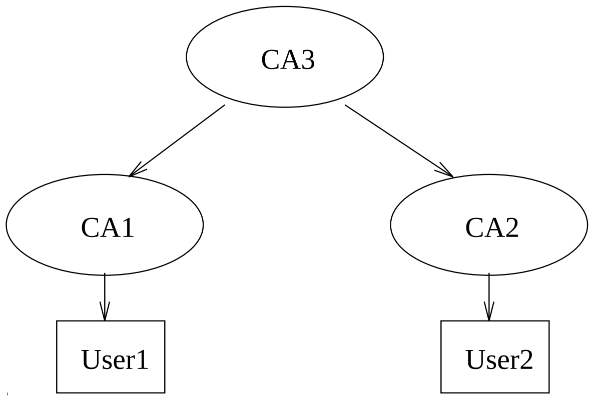



Figure 11.10: Hierarchical Model for Certification Authorities.

- the period during which the certificate is valid,
- a version number,
- a serial number.

When a CA issues a certificate to a user, the CA provides a copy of its public key. With this key the user can validate certificates issued to other users subscribing to the same authority. To validate certificates issued by different authorities, a *certification path* (i.e., a path of trusted certificates) must be constructed between the two users. For example, if *User1* subscribes to *CA1*, *User2* subscribes to *CA2*, and both *CA1* and *CA2* have a trusted relationship with *CA3*, the certification path *(C1, C3, C2)* can be constructed, allowing *User1* and *User2* to obtain each other's public key. This concept of CAs generating certificates for other CAs is called *cross certification.* Figure 11.10 illustrates a hierarchical CA structure for cross certification. In the hierarchical model, CAs only generate certificates for the entities (i.e., CAs or users) below them.

An asymmetric key management scheme is presented in the directory system authentication framework, described in Recommendation X.509 [CCI88d]. The directory can be used to store public key certificates for MHS entities. These certificates can be accessed by other MHS entities to compute and/or validate the integrity and confidentiality of MHS messages.

Within the MHS, public key certificates may be conveyed by several methods. When an MHS entity initiates a connection to a peer, it may transfer its public key certificate for the peer to use to validate its credentials. Certificates can be registered with MHS entities, and transferred in MHS messages and reports. MHS entities can also obtain public key certificates by some means outside the MHS, such as accessing the X.500 directory.

An important aspect of key management is the management of certificate revocation. Certificates may be revoked for a number of reasons including:

- a private key is compromised,
- a user's change of affiliation,

- the lifetime of a certificate expires.

Lists of revoked certificates are called Certificate Revocation Lists. An entry on this list dissolves the binding between a user's identity and his public key and includes a time stamp indicating at what time the dissolution occurred.

## 11.6 X.400 Services

This section presents the MHS security features that counter the vulnerabilities described in Section 11.4. The features can be categorized into three general services. First, they allow various MHS entities to authenticate their identity. Second, they protect messages against modification, and third, they protect messages against unauthorized disclosure.

The security features provided by the MHS apply only to messages submitted directly to an MTA by an MTS user (i.e., a User Agent, Message Store, or Access Unit). They do not apply to communication between the MHS user (e.g., a person) and the MHS (e.g., the person's UA). Thus, the scope of MHS security services extends, for example, to communication between two UAs, but not to communication between two people.

Many of the MHS security services require security capabilities within the UA, but not the MTA. For example, to ensure the confidentiality of a message, the originating UA encrypts the message content, and submits the message to the MTS. Any MTA that handles the message uses envelope information to make decisions (e.g., routing), never needing access to the message content. Some security services, however, require MTAs with security capabilities. For example, to ensure that an originating MTA submitted a message for delivery, the originating MTA must generate and return proof of the submission to the originating user. Some of the MHS security services apply to the MS (Message Store) as well as to UAs and MTAs, such as services involving the exchange of security labels. In general, however, the MS is transparent to security features that apply between the originating and recipient UAs.

Many of the MHS security services rely on encryption techniques. Most services are flexible regarding whether asymmetric or symmetric encryption techniques are used, and more specifically, which algorithms are used. Some services, such as the non-repudiation services, require an asymmetric encryption algorithm.

The remainder of this section describes specific MHS security services. These services include: *message security labelling, secure access management, origin authentication, data integrity, data confidentiality, non-repudiation,* and *security management.*

### 11.6.1 Message Security Labelling

The *message security labelling* service binds a security label to an MHS object. Security labels can be bound to transferable objects (e.g., messages, reports), MHS entities (e.g., MTAs, UAs), and associations between peer MHS entities (e.g., UAs and MTAs, MTAs and MTAs).

The following simple example illustrates a use of security labels. An MTA services two users: *User1* and *User2*. Each user registers a set of security labels with the MTA. At some later time, a message addressed to both users arrives at the MTA. The message contains a security label with a security classification of *Secret*. The MTA examines the security labels registered for its users, and ascertains that *User1* has registered a *Secret* security label, however, *User2* has not. Depending on the security policy in force, the MTA may deliver the message to *User1*, and non-deliver the message to *User2*. By the same mechanism, the MTA may prevent *User2* from originating messages containing security labels classified as *Secret*.

As illustrated in the example, security labels can be used to control the sensitivity of messages originated by and delivered to a user. Section 11.6.2 describes how security labels can also be used to prevent the misrouting of messages.

## 11.6.2 Secure Access Management

*Secure access management* enables authentication between peer entities in the MHS. It counters the threats of masquerade, misrouting, and the replay of connection requests. *Secure access management* can be divided into two components: *peer entity authentication* and *security context*.

### Peer Entity Authentication

The *peer entity authentication* service allows two adjacent components in the MHS to create a secure association by transferring authentication credentials. For example, a UA may provide a password to its MTA when establishing a connection to submit a message. This service counters the threat of masquerade (i.e., impersonation of one MHS entity to another).

To provide *peer entity authentication*, the connection initiator transfers either simple authentication credentials (i.e., passwords) or strong authentication credentials (i.e., signatures) to the connection recipient. If strong authentication is used, the signature is applied to an asymmetric token, called a *bind* token.

In the bind token's signed-data, the connection initiator places a non-repeating number. This number allows the recipient to detect replay threats. For example, if a recipient receives a connection request where the token's non-repeating number duplicates a number received previously, the recipient can assume that the connection request is a replay of the previous connection.

The initiator may use the bind token's encrypted-data to transfer secret information, such as a symmetric encryption key. The communicating parties can use this key to encrypt data transferred across the connection (see sec. 11.6.3 for an example).

The connection initiator signs the bind token using its private key. The recipient validates the token signature using the initiator's public key certificate. This certificate may be registered with the recipient, transferred during the authentication process, or distributed by some other means.

### Security Context

During the peer authentication process, the initiator may propose a *security context.* A security context is a set of security labels which can determine the sensitivity of messages passed over an association. If the initiator has registered a set of security labels with the connection recipient, the proposed security context must be a subset of the registered labels.

The following examples illustrate how security contexts control the transfer of messages over an association, and, in specific instances, counter the threat of message misrouting. If a security context is established between an originator and the originating MTA, the originator may only be allowed to submit messages with security labels permitted by the security context. If a security context is established between two MTA's, the transfer of messages and reports may be determined by the security label of the message or report, and the security context. This allows security labels to be used for routing purposes; only trusted MTAs (i.e., MTAs capable of establishing a security context) will be used to route a message. If a security context is established between a recipient and the delivering MTA, the MTA may only be allowed to deliver messages and reports with security labels permitted by the security context. If the security label for a message is allowed by the recipient's registered security labels, but not by the recipient's current security context, the MTA may retain the message for delivery at a later time.

## 11.6.3 Origin Authentication

*Origin authentication* is a set of security services allowing communicating parties to authenticate their identities. It comprises the following services: *message origin authentication, report origin authentication, proof of submission,* and *proof of delivery.*

### Message Origin Authentication

*Message origin authentication* allows the identity of a message originator to be verified. This service counters the threat of masquerade (i.e., impersonation of the message originator). Since origin authentication has limited utility without content integrity, the *message origin authentication* service also provides assurance that the message content has not been modified. If a security label is present in the message, this service also enables proof of association between the security label and the message. *Message origin authentication* can be provided by one of two methods: a message origin authentication check or a content integrity check.

The message origin authentication check allows the identity of a message originator to be verified by the message recipient(s), and any MTA transferring the message. It is provided on a per-message basis using an asymmetric encryption technique.

The message origin authentication check is a digital signature included in the message envelope. The originator computes the signature as a function of the message content, the message content identifier (an optional attribute generated by the originator to facilitate the correlation of a message with any reports it may provoke), and the message security label. If the message content is encrypted, the signature is computed as a function of the encrypted content. Thus, the identity of the originator can be confirmed without the need to see the plaintext content.

If the signature is computed using the plaintext content, the message origin authentication check also provides *non-repudiation of origin* (see Section 11.6.6). This provision is not maintained if the signature is computed using the encrypted message content. The message originator, although unable to deny sending the encrypted content, can deny that the content decrypted by the recipient is the same as the original plaintext content.

The message origin authentication check is computed using the originator's private key. The service places no restrictions on the originator regarding which asymmetric algorithm is used. The originator conveys the object identifier for the algorithm, any input parameters required by the algorithm, and the signature generated by the algorithm, in the message envelope.

The message recipient(s), and any MTA transferring the message, can validate the signature using the originator's public key certificate. This certificate may be transferred in the message envelope, or obtained by some other means.

The second method to provide *message origin authentication* is the content integrity check. The content integrity check allows the identity of the originator to be verified by the message recipient(s), and possibly by any MTA transferring the message. It is provided on a per-recipient basis, using either symmetric or asymmetric encryption techniques.

The content integrity check is a cryptographic checksum included as a per-recipient field in the message envelope, or in the message token. A distinct token can be generated for each message recipient. If the secrecy of the check is required, the originator places it in the token's encrypted-data. Unlike the message origin authentication check, the content integrity check must be computed as a function of the plaintext message content.

The originator may choose either a symmetric or asymmetric encryption algorithm to compute the check. If the originator chooses a symmetric encryption algorithm, a symmetric encryption key is used by the message originator to compute the check, and by the message recipient(s) to validate the check. This key can be transferred in the token's encrypted-data, or distributed by some other means (e.g., by prior agreement). Since only the originator and the recipient(s) share this key, no MTA transferring the message can authenticate the message.

If the content integrity check is computed with an asymmetric encryption algorithm (i.e., is a digital signature), the originator's private key is used to generate the check. The recipient validates the check using the originator's public key certificate. This certificate may be transferred in the message envelope, or obtained by some other means. Providing the originator does not transfer the check in the token's encrypted-data, any MTA handling the message can validate the check.

The content integrity check can be computed using any symmetric or asymmetric algorithm understood by both the originator and the recipient. All information relevant to the algorithm can be conveyed with the check.

If an asymmetric algorithm is used to compute the check, *non-repudiation of origin* (see Section 11.6.6) is provided in addition to *origin authentication*. If a symmetric algorithm is used, *non-repudiation of origin* can be provided by placing the content integrity check in the token's signed-data or encrypted-data. This is because the check is computed using the plaintext content, then signed by the originator. The originator may also transfer a security label with the content integrity check in either the token's signed-data or encrypted-data, to bind the security label to the message content.

### Report Origin Authentication

The *report origin authentication* service enables a message originator to authenticate the origin of a delivery/non-delivery report. This service counters the threat of masquerade (i.e., impersonation of the report originator). It is provided to the message originator, as well as to any MTA transferring the report, on a per-report basis using an asymmetric encryption technique. If a security label is present in the report, the service binds the security label to the report.

The reporting MTA provides this service by generating a report origin authentication check (i.e., a digital signature) and sending it in the report. The report origin authentication check may be generated when the message origin authentication check is present in the subject message. The report signature is computed as a function of the content identifier and security label of the subject message, the name of the recipient, and for:

**a delivery report:** the time the message was delivered, and if requested by the originator, *proof of delivery* (see Section 11.6.3),

**a non-delivery report:** the reason and diagnostic for non-delivery.

The report origin authentication check is derived using the reporting MTA's private key. The check is validated by the originator of the subject message, and any MTA transferring the message, using the MTA's public key certificate. This certificate may be transferred in the report, or obtained by some other means.

### Proof of Submission

*Proof of submission* allows a message originator to obtain proof that its MTA submitted a message for delivery to the intended recipient(s). This service counters the threat of masquerade (i.e., impersonation of an MTA to an MTS user). It is provided on a per-message basis using symmetric or asymmetric encryption techniques.

The message originator requests the service by submitting a *proof of submission* request with a message. The originator's MTA returns the proof in an acknowledgment. The proof is computed as a function of the submitted message arguments (i.e., the submitted message without the content), the message identifier (which is added by the MTA), and the time the message was submitted.

To generate the proof using an asymmetric encryption algorithm, the MTA signs the acknowledgment using its private key. The message originator validates the signature using the MTA's public key certificate. This certificate may be registered with the originator, transferred in the acknowledgment, or obtained by some other means. An asymmetric *proof of submission* also provides *non-repudiation of submission* (see sec. 11.6.6).

If the message originator transferred a symmetric encryption key to the MTA during the authentication process (see sec. 11.6.2), the MTA can compute the *proof of submission* using this key. A symmetric *proof of submission* does not provide *non-repudiation of submission.*

#### Proof of Delivery

*Proof of delivery* allows a message originator to verify that a message was delivered to the intended recipient(s). This service counters the threat of falsely acknowledged receipt. It is provided on a per-recipient basis using symmetric or asymmetric encryption techniques.

The message originator requests the service from each message recipient (i.e., proof may be requested from some recipients, but not from others). The recipient returns the proof in a delivery report. Although the report is created by the delivering MTA, the proof included in the report is generated by the recipient MTS user (e.g., the recipient UA). The proof is computed as a function of the recipient's name, the time the message was delivered, and the following information from the subject message: the content identifier, the security label, and the plaintext content.

To generate the proof using an asymmetric encryption algorithm, the recipient signs the report using its private key. The message originator validates the signature using the recipient's public key certificate. This certificate may be transferred in the report, or obtained by some other means. An asymmetric *proof of delivery* also provides *non-repudiation of delivery* (see sec. 11.6.6).

If a symmetric encryption algorithm is used, the recipient computes the *proof of delivery* using a symmetric encryption key. The originator uses the same key to validate the proof. The X.400 Recommendations do not define how this key is distributed between the communicating parties. A symmetric *proof of delivery* does not provide *non-repudiation of delivery.*

A message recipient is not mandated to return *proof of delivery.* That is, even if the originator requests the service, the recipient has the option of not returning the proof. Thus, not receiving *proof of delivery* does not imply non-delivery of the subject message.

### 11.6.4 Data Integrity

*Data integrity* is a set of security services verifying that the content of a message has not been modified, and if a sequence of messages is transferred, that the sequence has been preserved. It comprises the *content integrity* and *message sequence integrity* services.

#### Content Integrity

The *content integrity* service allows an originator to provide proof that the content of a single message has not been modified. As mentioned previously, *content integrity* is meaningless to a user without *origin authentication. Content integrity*, on its own, authenticates the content of a message; however, the message may have been submitted by an imposter. *Origin authentication*, on its own, verifies the identity of message originator; however, the content received may not match the content originated. Thus, *content integrity* and *origin authentication* are provided by the same security mechanisms. These mechanisms, the content integrity check and the message origin authentication check, are described in section 11.6.3.

#### Message Sequence Integrity

*Message sequence integrity* provides proof that the ordering of a sequence of messages sent from an originator to a recipient has been preserved. This service counters message sequencing threats, such as the replaying and re-ordering of messages. It is provided on a per-recipient basis using symmetric or asymmetric encryption techniques.

To provide the service, the message originator generates a sequence number, which identifies the position of the message in the sequence. This number is transferred in the message token's signed-data, or if the secrecy of the number is required, in the token's encrypted-data. Each originator/recipient pair using this service maintains a distinct pair of sequence numbers. One drawback with the *message sequence integrity* service is that it requires all users to maintain pairwise sequence numbers with (potentially) all other users.

### 11.6.5 Data Confidentiality

*Data confidentiality* is a set of services used to protect data against unauthorized disclosure. It comprises the *content confidentiality* and *message flow confidentiality* services.

#### Content Confidentiality

*Content confidentiality* prevents the disclosure of the plaintext content of a message to any party other than the intended recipient(s). It is provided on a per-message basis using an asymmetric or symmetric encryption technique. The encrypted content is unintelligible to any MTA handling the message.

If the originator chooses an asymmetric algorithm, the recipient's public key is used to encrypt the message content. The recipient uses its private key to decrypt the content. If an asymmetric encryption algorithm is used, the message can only be addressed to a single recipient (i.e., the recipient whose private key is paired with the public key used to perform the encryption).

If the originator chooses a symmetric algorithm, delivery to multiple recipients is possible. The originator encrypts the content using a symmetric encryption key. This key may be distributed to each message recipient by placing the key in the encrypted-data of the message token for that recipient. The key may also be distributed by some other means (e.g., by prior agreement).

The message originator can encrypt the content using any symmetric or asymmetric algorithm understood by both the originator and the recipient. All information relevant to the algorithm, such as the algorithm's object identifier and any input parameters, can be conveyed in the message envelope or the signed-data of the message token.

**Message Flow Confidentiality**

*Message flow confidentiality* allows the message originator to conceal the flow of a message through the MHS, protecting against information that may be derived from its observation. This service counters the threats of traffic analysis and loss of anonymity of the communicating parties. It is provided by a technique called *double enveloping.*

To provide this service, the message originator specifies that the content of a message is itself a complete message (usually encrypted). The recipient on the outer envelope, upon receiving the message, forwards the message to the recipient named on the inner envelope. Double enveloping only provides a limited *message flow confidentiality* service. A more comprehensive service would include *traffic padding* and *routing control*, which are outside the scope of the X.400 Recommendations.

### 11.6.6 Non-repudiation

Non-repudiation services provide unforgeable evidence that a specific action occurred. The MHS provides the following non-repudiation services: *non-repudiation of origin*, *non-repudiation of submission*, and *non-repudiation of delivery*. *Non-repudiation of origin* protects against any attempt by a message originator to deny sending a message. *Non-repudiation of submission* protects against any attempt by an MTA to deny that a message was submitted for delivery. *Non-repudiation of delivery* protects against any attempt by a message recipient to deny receiving a message.

The non-repudiation services are similar to their weaker proof counterparts (i.e., *proof of submission, proof of delivery,* and *message origin authentication*); however, non-repudiation provides stronger protection, because the proof can be demonstrated to a third party. Digital signatures are used to provide non-repudiation. For example, if a recipient returns *proof of delivery* by signing a report, *non-repudiation of delivery* is also provided. Since only the recipient's private key could have generated the signature, the signature provides unforgeable evidence of message delivery. Symmetric encryption cannot guarantee non-repudiation. Since both the originator and recipient share the symmetric encryption key, either party can generate the proof.

The exact mechanisms used to provide *non-repudiation of origin, non-repudiation of submission*, and *non-repudiation of delivery* are described in Section 11.6.3. Non-repudiation services may also be provided by a third party notary; however, third party notaries are outside the scope of the X.400 Recommendations.

### 11.6.7 Security Management

Throughout this section, references have been made to MHS *security management* services, namely, the *registration of security labels*, and the *registration of credentials*. For completeness, these services are described in this section.

The *registration of security labels* service allows an MTS user to convey a set of security labels that are maintained by its MTA. The MTA uses these labels primarily to control the delivery of messages to the MTS user.

The *registration of credentials* service allows peer MHS entities (i.e., an MTS user and its MTA) to register credentials with each other. For simple authentication, credentials comprise the password associated with the entity. For strong authentication, credentials comprise the entity's public key certificate. These credentials are used primarily when the peer entities are establishing an association.

## 11.7 X.400 Security Limitations

The previous section described specific X.400 services that counter threats to the MHS. Although the services employ a broad scope of security mechanisms, there are some limitations to the 1988 X.400 security architecture. These limitations pertain to the token, the message store, and some services provided by the MTS that access the content of messages.

One security limitation is that data in the token is encrypted before it is signed. This is considered a bad practice, because the recipient can only authenticate the encrypted data, not the plaintext data. In a worst case scenario, a malicious party can intercept a message, and create a new message keeping the encrypted content of the original message, but generating a new message token. The new token would be signed by the malicious party. Under these circumstances, the message recipient could be fooled into believing that the malicious party originated the message. It should be noted that depending on how the security services are implemented, this scenario can be avoided.

A second limitation pertains to the MS (Message Store). An MS can accept the delivery of messages on behalf of a UA. If a message originator requests *proof of delivery* for a message whose content is encrypted, and the message is delivered to an MS, the MS would require access to the encryption key to provide the service (the proof must be computed using the plaintext content). This would involve providing the MS with the recipient's private key, or the symmetric key shared between the originator and the recipient. Neither case is desirable from a security standpoint. This is one scenario where a recipient (i.e., the MS) might ignore the originator's request for *proof of delivery*.

A final limitation pertains to MTS services which access the message content or message recipient(s). An MTA may perform conversion on incoming messages, such as converting telex data to IA5 data. Any type of conversion invalidates integrity checks. Also, if the content is encrypted, the conversion cannot be performed without the MTA first decrypting the content. To decrypt the content, the MTA would require access to the encryption key, which is not desirable from a security standpoint. Similar problems result from services that modify the message recipient(s), such as an MTA expanding a distribution list or redirecting a message. If an MTA performs such a service on a message where the recipient's public key is used as input to some security service (e.g., to encrypt the encrypted-data of a token), the security service must be recalculated using the public key of the new recipient(s).

# Chapter 12

# X.500 Directory Services

**Michael Ransom**

The standardized infrastructure of the Open Systems Interconnection (OSI) application layer includes the Directory, a specialized database system that can be used by other OSI applications, and by people, to obtain information about objects of interest in the OSI environment. Typical Directory objects correspond to systems, services, and people. Examples of information found in the Directory include telephone numbers, electronic mail addresses, postal addresses, network node addresses, public key identity certificates, and encrypted passwords. Because of existing and proposed privacy legislation such information, more often than not, is expected to be subject to various security policies that dictate how disclosure and modification are to be controlled. The Directory standard, as originally published in 1988, pointed out the need for a standardized access control mechanism, but did not include specifications for any particular mechanism. Since that time, the standards committee charged with maintenance of the Directory standard has been working to remove that deficiency as well as a number of others. This effort has culminated in the publication of a new edition of the Directory standard in 1993 that incorporates a series of amendments and one new part covering replication. For access control, there are four amendments that collectively describe two standardized access control mechanisms and improvements to the Directory Authentication Framework. The new access control mechanisms will be available on an optional basis in implementations of the new Directory standard.

This chapter focuses on the two standardized access control mechanisms and provides insight into their use by characterizing parts, or fragments, of security policy that can be easily supported. In addition, some important policies that are not supported are discussed. The primary goal is to help system administrators and security managers understand the general character of security policy requirements and authority relationships that can be accommodated by the new mechanisms.

The body of this chapter is organized into four major sections. The first provides a brief overview of the Directory system and identifies the general scope of policy issues that can be addressed using the standardized access control mechanisms. The second and third sections progress towards a more detailed explanation and characterization of policy elements that can be represented and enforced by the mechanisms. These sections begin by using popular security policy models to provide an overview of what the standardized access control mechanisms can and cannot control and what information is used by the mechanisms to make access decisions. Next, the Directory operations are reviewed to elucidate how access control relates to each. Some specific examples of controls for several operations are then considered in detail to show how access decision making works. The examples also provide a basis for building a taxonomy of supported policy encodings. The taxonomy is presented at the end of the third section. Finally, the fourth section characterizes some important policy issues that cannot be directly supported by the mechanisms.

## 12.1 Introduction to X.500

The Directory standard is a joint effort of the International Organization for Standardization / International Electrotechnical Commission (ISO/IEC) and The International Telephone and Telegraph Consultative Committee (CCITT). The standard is published jointly as ISO/IEC 9594 and as the CCITT X.500 series of recommendations. In general, the Directory adheres to a client/server paradigm, with the clients referred to as Directory User Agents (DUA) and the servers as Directory System Agents (DSA). This section provides a high–level overview of how the Directory is modeled in terms of architectural components and how the components relate to each other.

### The Information Model

This section describes the basic model of how information in the Directory is organized. The model defines terms for the units of information in the Directory; it also defines the relationships among the units. There are essentially three kinds of information held by the Directory:

1. *user information* that is intended for use primarily by the people and systems that access the Directory to obtain data such as electronic mail addresses, phone numbers, network node addresses, and public key identity certificates;

2. *operational information* that is intended for use primarily by the Directory system itself — examples of such information include access controls and internal consistency requirements that the Directory must maintain;

3. *server information* that is used by each server to identify the location and contents of other servers.

Server information is outside the scope of this introduction. The basic units of user information and operational information is illustrated in 12.1. The Directory database consists of a collection of entries each of which contains of one or more attributes. Each attribute, in turn, consists of a type and one or more values. Thus, an entry containing information about

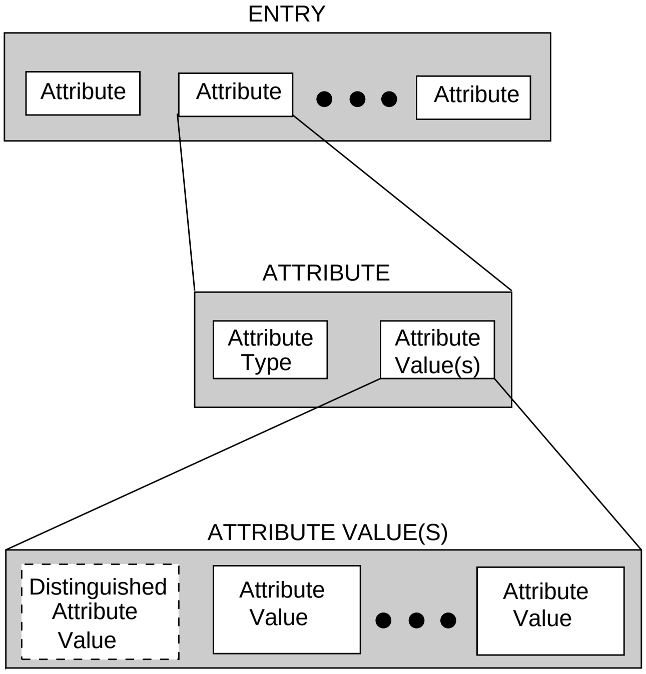



Figure 12.1: Structure of an entry.

Michael Ransom might contain attributes of type common name, surname, phone number, fax number, e-mail address, and public key certificate. Each entry must contain an attribute type called *object class* that defines the kind of real world object the entry represents; the object class for the example entry for Michael Ransom might have object class *organizational person*. Object classes are used to define the types of attributes that can appear in each entry; they can also be used in selecting entries during a Directory query operation. Typical object classes include people, computers, and software applications.

Each entry must also contain at least one attribute that is used in forming a name (i.e., access key) for the entry. The attribute value that is designated to participate in the name is called a *distinguished attribute value*. For the entry representing Michael Ransom, the name of the entry could be built using a value of the surname attribute or perhaps a value of the common name attribute. The attribute(s) to be used in naming entries of each object class is(are) defined by an administrative authority and enforced by the Directory.

Naming an entry in the Directory, however, involves more than just distinguishing attribute values in each entry. To facilitate the scalability of the database, entries are arranged into a tree structure such that each subtree can be assigned to different administrative authorities as needed when the database is world–wide. The tree structure is defined by the full name of each entry in the database. This means that each entry, in effect, inherits part of its name from the entries that are on the same branch and closer to the root of the tree. The administrators of a subtree are responsible for resolving naming conflicts within that subtree.

The structure of the tree is flexible but the branching points closest to the root are usually thought of as demarking a subtree for each country; under each country there is expected to be subtree branch points for organizations, organizational units, and localities. Entries representing people will most likely occur within the subtree for an organization, organizational unit, or locality. The tree is usually drawn upside-down with the root at the top of the drawing and leaf nodes at the bottom boundary. Figure 12.2 illustrates an example of the tree structure. In the example, the boxes represent entries in the tree. The middle box immediately below the root represents the entry for the United States; this entry is named using the attribute type COUNTRY with distinguished value US. Since this entry is immediately below the root, its full name is COUNTRY = US (abbreviated C=US).

The middle box immediately below the C=US entry represents the Department of Commerce (DoC) and is named using the attribute type ORGANIZATION with distinguished value DoC (abbreviated O = DoC). The full name of the DoC entry is made up of a combination of its distinguished attribute value and the names of all the entries above it on the same branch. The full name of the DoC entry is written { C = US, O = DoC }.

Similarly, below that entry is an entry representing NIST as an organizational unit of the DoC. It is named using the Organizational Unit attribute type with a distinguished value of NIST. The full name of the NIST entry is written { C = US, O = DoC, OU = NIST }.

Finally, below the NIST entry is the entry representing the person whose name is Michael Ransom. It is named using the Surname attribute type with a distinguished value of Ransom. The full name of this entry is written { C = US, O = DoC, OU = NIST, S = Ransom }.

The term *Directory Information Tree* is used to refer to the tree structure view of the Directory database.

### Model of the Directory as a Distributed Database System

The Directory is usually thought of as a distributed database system that is somewhat specialized. Roughly speaking, a database system is said to be *distributed* when the data is dispersed among several computers on a network and the computers cooperate over the network to provide a coherent database service to the user. The Directory is specialized in ways that allow it to be dispersed among computers that share a world–wide network. This section presents a simplified model of the distributed aspects of the Directory.

The architectural components of the Directory are illustrated in figure 12.3. Each Directory System Agent (DSA) holds a part of the database and also holds information about the location and contents of other DSAs. A user accesses the Directory through a Directory User Agent (DUA). Interaction among DUAs and DSAs is described later in this chapter.

## 12.2 Policy Aspects Supported by X.500 Access Control

From a design perspective, the foundation of Directory access controls is provided by a policy model known as the*access matrix model* which, in turn, is generally based on a simple table of

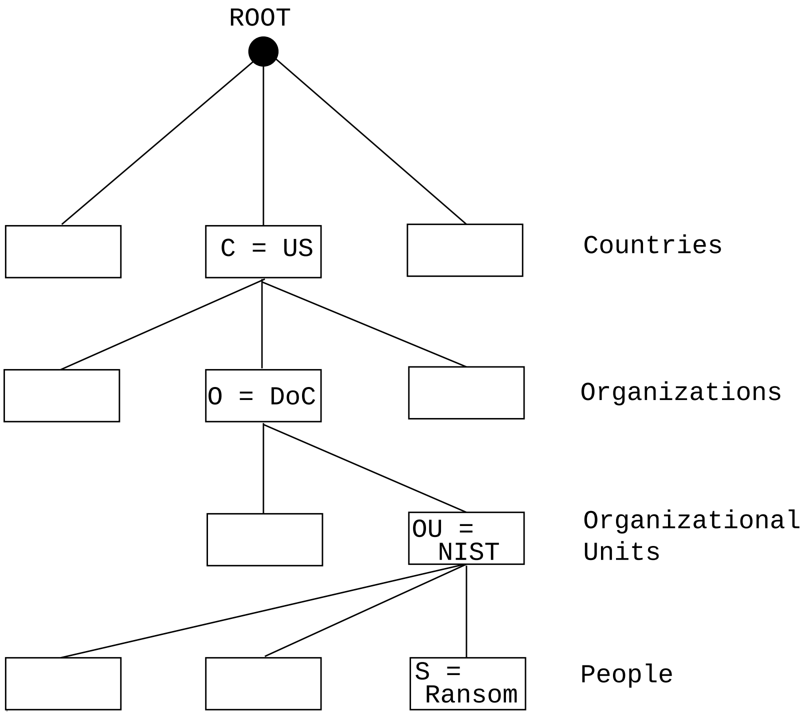



Figure 12.2: Example of the Directory Information Tree.

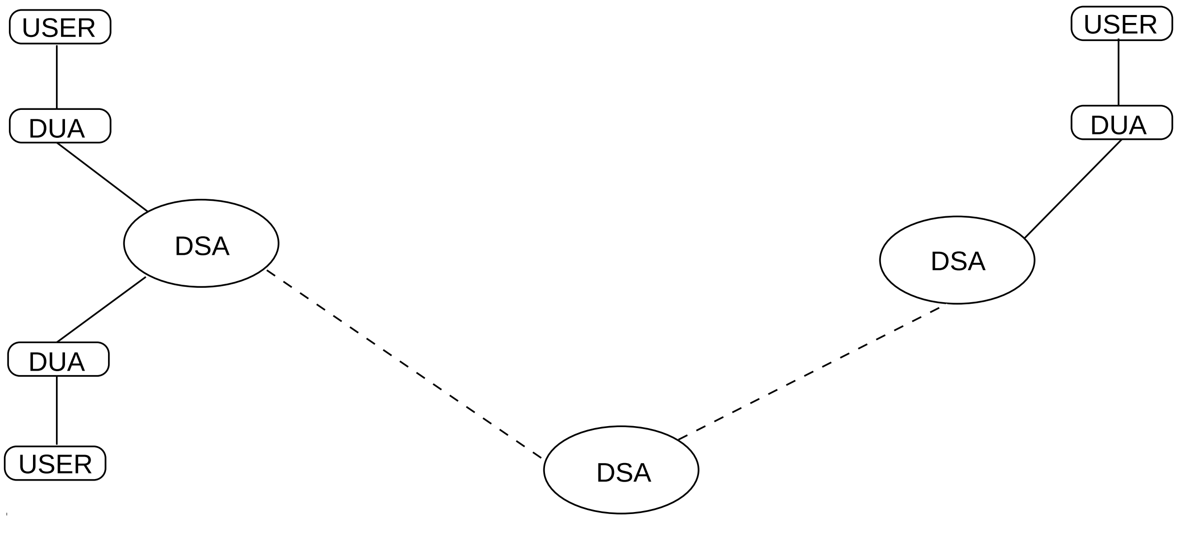



Figure 12.3: Components of the Directory.

rules relating who (what *subjects*) can do what (have what *rights*) to what (which *objects*). There are basically two approaches to expressing the rows and columns of this matrix in terms of access rules:

a) *capabilities*: associated with each user is a list of what rights the user has to what objects;

b) *access control list (ACL)*: associated with each protected object is a list of what users can exercise what rights to it.

The essential differences between capabilities–based schemes and ACL–based schemes can be illustrated by a simple example. Suppose a large but attendance restricted conference is being planned and the conference organizers are considering ways to control who gets into what sessions. Conference registration entitles the attendee to a certain track of sessions.

In a capabilities–based scheme, each registered attendee would be given a special badge that indicates what sessions that user is entitled to enter, assuming the user has proper identification. The badges have a special logo, probably to make counterfeiting a little difficult, and each is numbered. When entering a session, the user shows the badge and identification to the guard at the door. The guard does not know in advance who is permitted to enter, and, indeed, it may be possible that there are no lists of attendees cross–referenced by session.

In a pure ACL scheme, the guards are each supplied with a list of which specific users are authorized to enter their session. The users may be issued badges publicizing their name, perhaps, but possession of a badge is not used as the basis for authorization nor identification. Note that there may be no need for corresponding lists of all of the sessions a particular user might be able to enter.

The new Directory standardized access control mechanisms support an ACL based approach, but not capabilities. The DSA plays the role of the guard, making decisions based on a user's identity and ACL information that is closely associated with the protected object. It is interesting to note that even though the standardized access control mechanisms for the new Directory use the ACL policy model, they do not directly support situations where the DSA needs to remember what has happened in the past. It is important, therefore, to understand not only that Directory access control is expressed in terms of ACLs, but also that certain ACL situations cannot be enforced using only the standardized mechanisms. The next section characterizes many of the aspects of access control policy that can be enforced using the standardized access control mechanisms for the new Directory. A later section characterizes some aspects that are not supported by those mechanisms. It is possible that future amendments to the Directory standard will provide capability–based access control. Capability–based access control is not standardized in implementations of the 1993 edition of the standard.

This section characterizes many of the important aspects of security policy that are directly supported by the new Directory access control mechanisms. From a policy perspective, the mechanisms can be used to enforce a wide range of security authority relationships where each authority defines and maintains ACLs for the protected objects that are under its control. When making an access control decision, the Directory considers all of the ACLs from all of the authorities that may influence that decision. The automated guard uses the ACLs to make a decision that is consistent with the relationship established among the relevant authorities.

This section first focuses on a simplified authority scenario where there is only one authority for the entire Directory Information Base (DIB). After exploring the flexibility of access control policy for a single authority, the discussion moves to more complicated authority scenarios involving multiple autonomous authorities and various forms of delegation of authority.

### 12.2.1 Scenarios Involving a Single Authority

The easiest way to begin an analysis of supported policy is to avoid the complications that arise when more than one security authority is considered. Suppose the entire DIB is managed by one organization which has a single security manager that is responsible for all facets of security policy. Furthermore, suppose the security manager has not delegated his authority in any way. In particular, there is no discretionary setting of access rights by anyone other than this manager. In this simplified authority environment, we can explore and characterize many of the access control rules that can be enforced using the standardized access control mechanisms. It should be noted that, in this simplified authority scenario, there is no essential difference between the two new Directory access control mechanisms; the major difference is only apparent when delegation of authority is considered. The difference is emphasized in a later section where scenarios involving multiple authorities are discussed.

A first characterization of supported security policy (in the simplified authority environment) views all access control in terms of three broad categories of policy:

- *Disclosure* — controlling release of information from the DIB;
- *Modification* — assuring that the DIB is changed only in a specified and authorized manner; and
- *Resource control* — controlling who has access to computing or communication resources (e.g., a DSA).

Each of these categories is expanded into specific policy issues below.

#### Disclosure Policy

A disclosure, or confidentiality, policy, in the context of DIB access control, essentially addresses control of information revealed to the requestor of a Directory operation. The Directory supports four types of query operation: READ, COMPARE, LIST, and SEARCH. There are also four types of modify operation: ADD–ENTRY, REMOVE–ENTRY, MODIFY–ENTRY, and MODIFY–DISTINGUISHED–NAME. This section briefly reviews the operations and discusses how confidentiality policy applies to each. This section also provides some specific examples of confidentiality policy fragments that can be supported.

The READ operation is used to extract the contents of a single entry whose name is specified in the request. It may also be used to verify the existence of a particular entry without returning any of the entry's content.

The COMPARE operation is used to compare an attribute value supplied in the request with the value(s) present in an entry whose name is also supplied in the request.

The LIST operation is used to obtain the names of the immediate subordinates of an entry whose name is specified in the request. The term "immediate subordinates" refers to the Directory Information Tree (DIT) view of the Directory. The immediate subordinates of a parent entry are all subordinate entries of that parent that are exactly one level below the parent in the DIT. It is important to note that LIST is designed to return entry names that were, presumably, unknown to the requestor prior to the operation.

The SEARCH operation is also designed to return entry names that are unknown to the user. SEARCH, however, can be used to find the names of all subordinates of a particular parent entry. SEARCH can also be used to extract the contents of the subordinates on a selective basis.

ADD–ENTRY is used to add a new leaf entry to the DIT; the operation request specifies the name of the entry to be added together with the attribute types and values that the new entry contains.

REMOVE–ENTRY is used to remove an entire leaf entry from the DIT.

MODIFY–ENTRY is used to perform a series of one or more modifications to a single entry. The kinds of modifications that may be requested include adding/removing an attribute, adding/removing an attribute value, replacing an attribute value, and modifying an alias.

MODIFY–DISTINGUISHED–NAME is used to modify the Relative Distinguished Name (RDN) or any component of the Distinguished Name of an entry. It also (indirectly) has the effect of changing the Distinguished Name of any entry that is subordinate to the entry being renamed. It may also have the effect of moving an entry (and all its subordinates) to another area of the DIT. This operation is another new feature of the new Directory; it replaces the less powerful MODIFY–RELATIVE–DISTINGUISHED–NAME operation in the 1988 Directory standard.

Directory operations, in general, may either succeed or result in one of several possible error conditions. When an operation succeeds, an "operation result" is returned to the requestor that contains a standardized collection of information. In some cases, operation results convey no information other than success of the operation. When an operation fails, an "error result" is returned to the requestor, indicating what error occurred. An error result may also convey some relevant diagnostic information (perhaps including DIB information).

Controlling disclosure of DIB information during query operations involves controlling several categories of information conveyed in operation results or operation errors. Specifically, the standardized access control mechanisms address the following categories of information:

- the contents of an entry (i.e., attribute types and attribute values) revealed in operation results;
- the contents of an entry revealed in error results;
- Distinguished Name of the entry (or entries) providing information conveyed in the result of a query operation;
- Distinguished Names revealed in error results;
- the contents of an entry used by the SEARCH operation to determine if that entry is to be used in formulating the operation result.

For each type of operation, the standardized access control mechanisms can control confidentiality for each applicable category of information independently of controls for other operation types. For a given operation type, the mechanisms can control information revealed in the operation result independently of controls on the same information when revealed in an error result.

Also, the categories distinguish between information held in an entry and the Distinguished Name of the entry; the Distinguished Name of an entry is not considered to be contained in the entry. Therefore, the contents of a particular entry and the name of that entry may be controlled independently for a given operation. These controls for different operation types are also independent of each other.

Similarly, the use of entry contents in the selection phase of a SEARCH operation can be controlled independently of controls on disclosure of the same information in SEARCH, READ and COMPARE operation results. During the selection phase of a SEARCH, the Directory checks each entry in the scope of the search to determine if it meets selection criteria specified in the request. If the entry satisfies the criteria, it is included in the SEARCH result, otherwise, it is ignored. For each selection criterion, the Directory checks confidentiality policy to determine if the requestor is allowed access to entry contents needed to evaluate the criterion; if access is denied, the criterion fails. This feature, for example, can be used to preclude inversion of a phone directory that is held in the DIB. The security manager may want to allow users to access phone numbers via the READ operation or the SEARCH operation while also denying the ability to perform a SEARCH operation where the selection criteria are based on a phone number. A SEARCH operation using selection on a phone number could be used to find the name associated with a given phone number. The standardized access control mechanisms allow the manager to specify that the phone number is accessible via READ or SEARCH but SEARCH cannot be used to find the (unknown) name associated with a known number.

The modify operations also potentially result in disclosure. Controlling disclosure of DIB information during a modify operation involves controlling only error results, since the operation results convey no information other than success of the operation. In general, the security manager may choose between error results in situations where modification policy denies a requested modification or where the modification is trying to add something that already exists. The options for each modify operation are summarized below. As for query operations, any error result which reveals a Distinguished Name is subject to confidentiality policy on that name.

If an ADD–ENTRY operation attempts to add an already existing entry, the security manager may choose to reveal the existence of the target entry, or he may choose to return an error that is intended to conceal the existence of the target entry.

If a REMOVE–ENTRY operation attempts to remove an existing entry, the Directory checks applicable modification policy to see if the requestor is allowed to remove the entry. When such policy denies a requested removal, the security manager may choose between returning an error result that is intended to avoid disclosure of the existence of the entry or an error result that does not protect disclosure of the existence of the entry.

Similarly, for each removal in a MODIFY–ENTRY operation, the Directory first checks applicable modification policy to see if the requestor is allowed to remove that particular item. When modification policy denies a requested removal of an existing attribute or value, the security manager may again choose to return an error result that is intended to conceal the existence of the item for which removal was denied; or, alternatively, he may choose to return an error that does not conceal its existence. For modifications that add an attribute or value, the Directory first checks to see if the item to be added already exists. If it does, the security manager may choose to return an error result intended to conceal the existence of the item or he may choose to return an error result that specifically reveals its existence.

Depending on the effect of a MODIFY–DISTINGUISHED–NAME operation, one or more modification policy checks are made to ensure the requestor has permission to perform the operation. If not, the security manager again has the option of either returning an error result intended to conceal the existence of the target entry or an error result that is not intended to protect its existence.

### Controlling Disclosure of Distinguished Names

As mentioned above, many of the operation results, and one error result, contain at least one entry name. Entry names may be the object of confidentiality policy because each name reveals information about the the structure of the DIT which may, in turn, reveal information about the organizational structure of the organization(s) that control(s) the name. For example, a private company may choose to have their subtree of the DIT reflect the company's true organizational structure while also regarding that structure as proprietary information. A company might want their DIT subtree to reflect the company's organizational structure because it helps employees use the Directory more effectively; they can use their knowledge of the organizational structure to find entries they need. A hypothetical policy might allow disclosure of Distinguished Names in query and error results generated for company "insiders" (i.e., employees of the company) while disallowing such disclosure in operation and error results generated for "outsiders." Such a policy is fully supported by the standardized access control mechanisms.

When confidentiality policy precludes disclosure of a Distinguished Name in an operation result, the Directory conceals the name by various means depending on what the operation is. For READ and COMPARE operation results, the Distinguished Name of the target entry is concealed by simply returning the same name that was specified by the user in the operation request. This action is also taken when avoiding disclosure of the Distinguished Name of the base entry for a LIST or SEARCH operation. Note that the name specified by the user in the operation request is either an alias name or the Distinguished Name; in either case, the Directory returns a name that was already known by the user.

Concealing the Distinguished Name of an entry immediately subordinate to the base of a LIST operation must be handled differently since the operation request does not provide a name that can be echoed back in the operation result. To conceal the Distinguished Name in this case, the Directory will take one of two actions. The responding Directory System Agent (DSA) first checks to see if a "locally defined alternate name" has been established. Such a name is "locally defined" because there is no standardized means of designating an "alternate name"; the alternate name is identified by the responding (i.e., local) DSA by means that are defined by the DSA implementor or by functional profiles. An alternate name is an alias name for the entry whose Distinguished Name is to be concealed. If an alternate name has been established in the responding DSA, the operation result will contain the alternate name. If an alternate name has not been established in the responding DSA, the entry is omitted from the operation result.

Similarly, for the SEARCH operation, the Distinguished Name of a nonbase object is concealed by using a locally defined alternate name if such a name is available. If an alternate name is not available, the entry is completely omitted from the operation result.

A particular error result, known as NAME–ERROR, contains an entry name that may be controlled by confidentiality policy. A NAME–ERROR result contains an entry name for which:

1. confidentiality policy allows the existence of the entry to be disclosed in an error result; and

2. confidentiality policy allows the disclosure of the name.

In the process of identifying such a name, several special cases arise that may involve returning an empty name or an alternate name. The use of alternate names is based on criteria similar to that described above for operation results.

### Modification Policy

Modification policy, in the context of DIB access control, is concerned with controlling the actions of modify operations. This section describes how modification controls apply to each modify operation.

For the ADD–ENTRY operation, modification policy can control whether or not a particular area of the DIT is allowed to receive new leaf entries. The general application of access control policy to an area of the DIT is discussed in a later section on "Encoding Policy in an ACL." If area permissions allow the new entry to be added, then the Directory makes additional modification policy checks for each attribute type and each attribute value that is to be contained by the new entry. If modification policy denies the addition of any of the proposed attributes or values, then the entire operation fails.

For REMOVE–ENTRY, modification policy can control whether or not an entire entry (including all of its contents) is allowed to be removed. Component attributes and values cannot be controlled independently with respect to the REMOVE–ENTRY operation (they may, however be independently controlled for the MODIFY–ENTRY operation as explained below). For each REMOVE–ENTRY operation, the Directory makes a single check of modification policy to see if the entire entry is allowed to be removed; there are no separate checks for each attribute and value inside the entry (as was the case for ADD–ENTRY).

In the case of MODIFY–ENTRY, the Directory first checks modification policy to see if the MODIFY–ENTRY operation may be used on the target entry. If so, for each attribute removal, the Directory makes one check of modification policy to see if the entire attribute (with all its values) can be removed. For each attribute value removal, the Directory makes one check of modification policy to see if that value can be removed (note that modification policy applicable to the attribute as a whole is not checked when the request is for removal of a particular value). For each attribute that is added, a check is made of modification policy to see if the attribute as a whole may be added; if so, a check of modification policy is made for each value to be added. Similarly, for each attribute value added to an existing attribute, a check of modification policy is made to ensure the new value may be added. Controls on an attribute as a whole are independent of controls on particular values of an attribute.

For MODIFY–DISTINGUISHED–NAME, the Directory first determines if the operation causes the target entry to "move" to a new immediate superior (parent) entry in the DIT. If the modification would result in the target entry having the same parent, then the Directory makes a single check of modification policy to determine if the renaming is allowed. If the modification would result in the target entry having a new parent, then the Directory makes two checks of modification policy: the first check determines if the entry (considered with the name it had prior to the operation) is allowed to be moved to a new parent; the second check determines if the DIT area that would be occupied by the moved entry (and all its subordinates) is allowed to receive moved entries. Control on renaming an entry without moving it to a new parent is independent of controls on whether or not an entry may be moved to a new parent.

### A Note on Security–Error

One of the error results that may be generated by the Directory is called a "Security–Error". This error result may be returned by any of the query or modify operations. When Security–Error is returned as a result of denial of access (because of confidentiality or modification policy), the Directory standard allows the Security–Error to contain one of two problem codes. The first option is to use a problem code which reveals that insufficient access rights caused the operation to fail. The other option is to use a problem code that gives no

| User | Permission | Protected Item |
|---|---|---|
| Ransom | grant Read | attribute type $X$ (in entry $Y$) |

Figure 12.4: Basic ACL.

information about what kind of security problem caused the operation to fail. Since the standard leaves this option open, it is anticipated that implementors of the new Directory will provide a way for the security manager to specify which option is to be used in each of the situations where it arises. Confidentiality policy should address which option is to be exercised in each particular situation.

### Encoding Policy in an ACL

Each fragment of security policy that is to be enforced by the standardized access control mechanisms must be expressed in terms of an ACL. A basic ACL has three components: one identifies which user the policy fragment applies to; another component identifies a particular access permission and whether that permission is granted or denied; the last component identifies what part of the Directory the ACL protects. A typical basic ACL, shown in figure 12.4, specifies the following confidentiality policy fragment: The user with surname "Ransom" is allowed to read (e.g., via the READ operation) all values of attribute type $X$ that are held in the entry with Distinguished Name $Y$.

In many cases it would be inconvenient, however, if security policy had to address each potential user and each protected item individually. To avoid these inconveniences, several types of collective controls are supported by the standardized access control mechanisms. More specifically, four levels of "collective controls" can be used to specify that a basic ACL applies to more than one user, more than one permission, and more than one protected item. The four levels of flexibility are:

1. an ACL may specify various collections of users to which it applies;

2. an ACL may specify various collections of protected items within an entry;

3. an ACL may apply to more than one permission category;

4. each ACL is assigned a scope of influence that defines the part of the DIT to which it applies.

The scope of influence of a particular ACL may be: a single entry; many entries that are related by virtue of their relative positions in the DIT and/or their object classes; or a dynamic "area" of the DIT that holds a subtree of existing entries and is allowed to grow via ADD–ENTRY and/or MODIFY–DISTINGUISHED–NAME.

The four levels of flexibility in defining collective controls may be used individually or may be arbitrarily combined to conveniently state a particular access control policy fragment.

| User Class | Permission | Protected Item | Scope of Influence |
|---|---|---|---|
| all users | grant Read | attribute type $X$ | all entries of object class $Y$ in subtree with root $Z$ |

Figure 12.5: ACL using collective controls.

Figure 12.5 is an example of a more flexible ACL that conveniently specifies the following confidentiality policy fragment: All users are allowed to read (e.g., via the READ operation) attribute $X$ in any entry with object class $Y$ within the DIT subtree that begins at the entry with Distinguished Name $Z$.

The example ACLs shown in figures 12.4 and 12.5 are not actually sufficient to enforce their respective confidentiality policy fragments using the standardized access control mechanisms. They are insufficient because for each query operation the standardized mechanisms first check to see if the requestor is allowed to apply the operation to any part of an entry that must be accessed to fulfill the operation. If so, the mechanisms then check to see if the requestor can access each specific attribute type needed to complete the operation. The mechanisms also perform a separate check to see if the requestor can access each value needed to complete the operation. The ACLs in figures 12.4 and 12.5 only specify permissions for a specific attribute type. They do not specify controls on the values of Attribute $X$, and they do not specify controls needed to allow the requestor to perform a query operation on any entry (regardless of what part of the entry is being read).

Now consider figure 12.6. Each row in the figure is considered to be a separate ACL. The first row provides the permission needed to allow all users to perform a READ operation (independent of what attributes are accessed by any particular instance of a READ). Without this permission, users would not be allowed to perform a READ operation on any entry (regardless of what attributes they are trying to read or how attribute types and values are controlled). The reason for these “entry–level” permissions will become apparent in the next figure.

The second row in figure 12.6 allows the query operation result to disclose a particular attribute type (i.e., type $X$). This row does not control any of the values associated with an instance of an attribute of type $X$; it only controls disclosure of the type information associated with the attribute.

The third row in figure 12.6 allows the query operation result to disclose any or all of the values contained in an attribute of type $X$. The standardized mechanisms also allow each individual value to be independently controlled.

It should be noted that the ACL representation used in the standardized access control mechanisms allows all three rows of figure 12.6 to be collapsed into a single row in which the third column specifies all three protected items. Collapsing the rows is possible because they all have the same information in the first, second, and fourth columns.

The first row of figure 12.6 is referred to as an “entry–level” control because the Protected Item is the entry as a whole. To illustrate the usefulness of entry–level controls, consider the

| User Class | Permission | Protected Item | Scope of Influence |
|---|---|---|---|
| all users | grant Read | entry | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant Read | attribute type $X$ | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant Read | all attribute values in an attribute of type $X$ | all entries of object class $Y$ in subtree with root $Z$ |

Figure 12.6: A more realistic set of ACLs.

following confidentiality policy fragment: All users are allowed to read (via the READ or SEARCH operations) attribute $X$ in any entry with object class $Y$ within the DIT subtree that begins at the entry with Distinguished Name $Z$. Suppose the policy fragment also states that attribute $X$ may not be used in SEARCH filter criteria. The ACLs needed to enforce the policy fragment are shown in figure 12.7.

The only difference between figures 12.6 and 12.7 is the second row in figure 12.7 which grants an entry–level permission called "Browse." This permission must be granted before any user is allowed to apply the SEARCH operation to any entry (regardless of controls on the attribute types and values for entries in the scope of a SEARCH). Note that the attribute type and value controls are the same for both READ and SEARCH; only the entry–level permissions are different.

Its interesting to analyze the policy differences between figures 12.6 and 12.7. The policy for figure 12.6 allows disclosure (to any user), via a READ operation result, of all the information associated with a particular attribute when that attribute is present in an entry of a particular object class within a particular subtree of the DIT. Since only the READ operation may be used, a user can only read attribute $X$ if he already knows the name of an entry that contains attribute $X$ (and which satisfies the object class and subtree location requirements). When requesting a READ operation, the user must specify a valid name for the entry to be read.

Using a SEARCH operation, however, a user may read the contents of an entry for which that user does not a priori know a valid name. In other words, the user may read entries that are not explicitly named in the operation request. If attribute $X$ contains phone numbers, then the policy in figure 12.7 would allow users to read phone numbers without first knowing a valid Directory Name for the object associated with the phone numbers (the object might be a person). Under the policy of figure 12.6, the user would have to know such a name before the phone numbers for that name could be read.

Another interesting facet of the policy expressed in figure 12.7 is that users may obtain attribute $X$ values in a SEARCH result, but they are not allowed to use attribute $X$ in a filter criterion of the SEARCH request. Use of attribute $X$ as a filter criterion is disallowed because there is no ACL that grants its use as a filter item.

| User Class | Permission | Protected Item | Scope of Influence |
|---|---|---|---|
| all users | grant Read | entry | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant Browse | attribute type $X$ | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant Read | attribute type $X$ | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant Read | all attribute values in an attribute of type $X$ | all entries of object class $Y$ in subtree with root $Z$ |

Figure 12.7: Allowing both READ and SEARCH.

Disallowing the use of attribute $X$ in a filter criterion precludes a user from searching a range of entries in an attempt to find the entry containing a specific value of attribute $X$. If, for example, attribute $X$ contains phone numbers then a user would not be allowed to search the Directory to find out what name is associated with a given number. As mentioned above, there are independent confidentiality controls on what may be used in a SEARCH filter criteria and what may be disclosed in a SEARCH operation result.

Figure 12.8 shows ACLs that enforce a policy that does allow inversion of telephone number information (assume attribute $X$ is telephone number).

In figure 12.8, two additional rows have been added to allow attribute type $X$ and the values of an instance of that attribute to be used in filter criteria within a SEARCH operation request. "FilterMatch" is the name of the permission category used by the standardized mechanisms to control what can be used in a SEARCH filter.

The differences between figures 12.7 and 12.8 also highlight an important principle that is always observed by the standardized access control mechanisms: the access control guard only grants access when there is an ACL that explicitly grants the required permission. Access is always denied when there is no ACL that explicitly grants the required permission; access is, of course, also denied when there is an ACL that explicitly denies the required permission.

The examples shown in figures 12.4 – 12.8 can be generalized into a taxonomy of access controls supported by the new Directory standardized access control mechanisms. The remainder of this section presents one such taxonomy. Each category in the taxonomy is shown in a box immediately preceding a description of the category. The statement of a particular policy fragment may freely utilize any useful combination of the categories.

**Subtree–dependent controls**

As previously mentioned, each ACL is assigned a scope of influence that identifies the

| User Class | Permission | Protected Item | Scope of Influence |
|---|---|---|---|
| all users | grant Read | entry | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant Browse | attribute type $X$ | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant FilterMatch | attribute type $X$ | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant FilterMatch | all attribute values in an attribute of type $X$ | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant Read | attribute type $X$ | all entries of object class $Y$ in subtree with root $Z$ |
| all users | grant Read | all attribute values in an attribute of type $X$ | all entries of object class $Y$ in subtree with root $Z$ |

Figure 12.8: Control of SEARCH filter.

part of the DIT to which it applies. The smallest scope of influence is a single entry; it is used when an ACL expresses policy that is applicable to a single entry in the DIT. When a policy applies to every entry in a subtree, it is not necessary to repeat the ACL for each of the entries. For convenience, it is possible to specify an entire subtree as the scope of influence for an ACL. A policy fragment with a subtree scope applies equally to each entry in the subtree and may be referred to as a *Subtree–dependent control*. Figure 12.6 is an example of Subtree–dependent controls.

If an organization chooses to build its part of the DIT such that there is a subtree for each organizational unit, then policy that applies to an entire unit could be easily expressed using ACLs whose scope of influence is defined to be a subtree.

Another important use of Subtree–dependent controls is to enforce policy regarding how the DIT is allowed to grow and change in shape. The ADD operation allows it to grow; the MODIFY–DISTINGUISHED–NAME operation allows entire subtrees to, in effect, be moved from one parent entry to another. Both of these operations may be controlled using Subtree–dependent access controls. When an attempt is made to add a new entry, the Directory first checks to see if there is a subtree ACL whose scope of influence includes the proposed entry's name. If there is such an ACL and it grants entry–level permission for the ADD operation, the Directory continues with the operation by checking modification policy to see if each proposed attribute type and value are allowed to be placed in the new entry. The controls on adding each type and value are also specified in ACLs whose scope of influence include the name of the proposed entry. If any one of the modification checks fails, the ADD operation fails (completely).

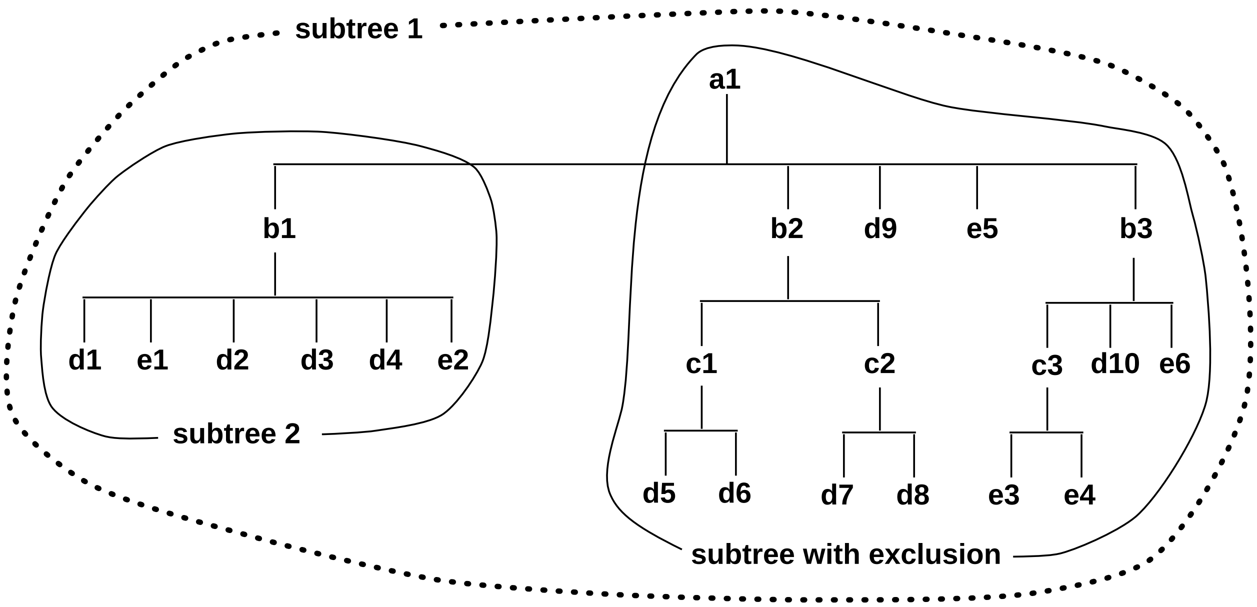



Figure 12.9: Subtree specifications.

Similarly, when a MODIFY–DISTINGUISHED–NAME operation has the effect of moving an entry to a new parent, the Directory first checks to see if there is a subtree ACL whose scope of influence contains the proposed new name of the entry to be moved. If there is such an ACL and it grants the appropriate entry–level permission to allow the operation to be applied, then the operation is allowed to succeed (subject to some additional checks described in the previous discussion of the MODIFY–DISTINGUISHED–NAME operation).

Finally, it should be noted that it is possible to define a scope of influence that excludes a specific branch of a subtree. For example, in figure 12.9, **subtree 1** and **subtree 2** are complete subtrees; the root of **subtree 1** is entry **a1** while the root of **subtree 2** is entry **b1**. Entry **a1** is also the root of the area labeled **subtree with exclusion** which is not a complete subtree because one of the branches (**subtree 2**) has been excluded. The ability to exclude specific branches from a subtree specification is an added dimension of flexibility in specifying ACL scope of influence.

## Class–dependent controls

For convenience, the standardized mechanisms also facilitate specification of a policy that applies to entries (in a subtree) that have a particular object class. Each entry in the Directory contains an attribute that specifies what object class it represents; some examples of object class are: *organization*; *organizational unit*; *person*; and *device*. A single ACL could express a policy that applies to a particular subtree but only affects entries of object class *person*. More generally, a single ACL could express a policy that applies to a boolean combination of object classes — for example, suppose a particular policy applied equally to entries of object class *organization* and to entries of object class *organizational unit.* A single ACL could be used to express the policy by assigning a scope of influence that includes *organization* **or** *organizational unit.* Combinations of object classes may be specified using the boolean operators ( **or**, **and**, **not**) to build elaborate scopes of influence.

An ACL that has a scope of influence defined in terms of object classes may be referred to as Class–dependent controls.

### Resource–dependent controls

Class–dependent controls which target object classes that are associated with resources other than information can also be regarded as being resource dependent. Examples of object classes that represent such resources include:

a) *device* — used to represent physical units which can communicate (e.g., modem, disk drive, computer);

b) *application process* — used to represent an element within an open system which performs the information processing for a particular application;

c) *application entity* — used to represent those aspects of an application process that are pertinent to OSI (e.g., network addresses).

Any object class that is derived from such object classes could also be regarded as representing a xxx information resource. For example, a subclass of *application entity* could be defined to represent Message Transfer Agents for electronic mail services. Since these object classes might contain network addressing information, it is possible that the Directory could be made partly responsible for controlling the use of resources by controlling access to network addresses. Security policy applying to the use of such resources and control of related network addresses could be regarded as either confidentiality policy or as resource control policy; when it is the latter, the associated ACL(s) may be referred to as *Resource–dependent controls.*

### Subtree Level–dependent controls

The scope of influence for an ACL can also be defined to include only a subrange of the levels in a particular subtree of the DIT. Such controls may be referred to as *Level–dependent controls.* Subtree levels are illustrated in figure 12.10 where, for example, the entries labeled b1 and e5 occupy subtree level 1 while e3 and d5 occupy level 3. Policy which applies equally to each and every entry in a particular level or subrange of levels can be expressed easily using features of the standardized mechanisms that are used to define ACL scope of

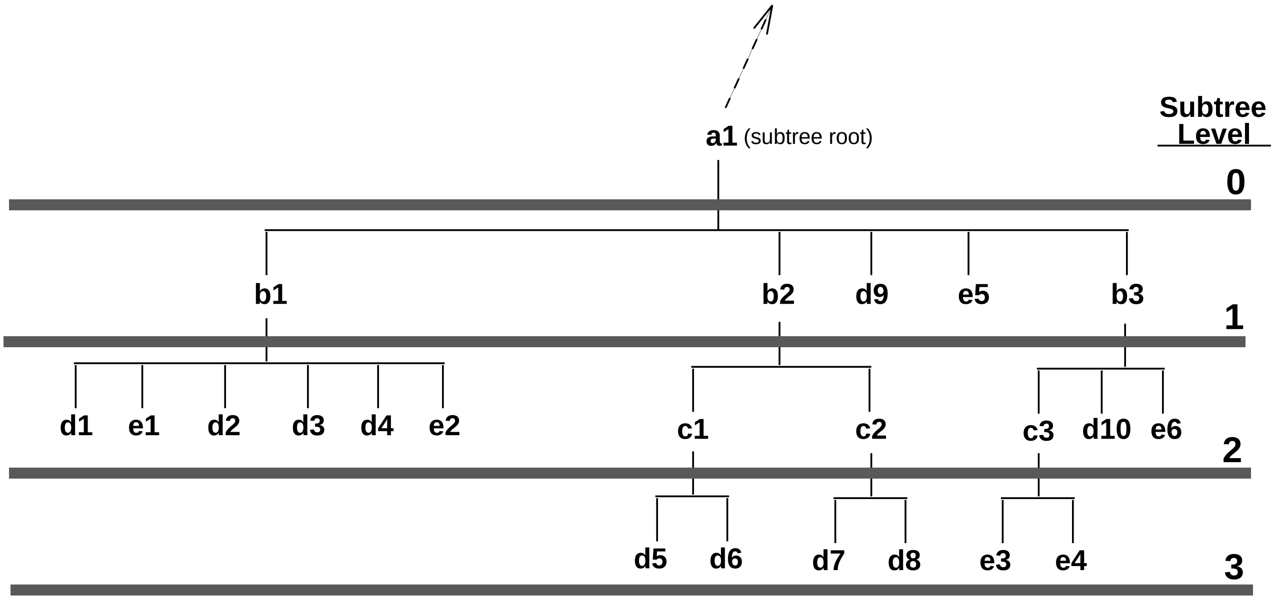



Figure 12.10: Subtree levels.

influence. All the levels in a valid subrange are contiguous (e.g., levels 1 and 3 do not form a subrange because level 2 is left out).

## Operation–dependent controls

In the discussion of figures 12.6 and 12.7 it was pointed out that each type of operation (e.g., ADD–ENTRY, READ) is at least partially controlled by an "entry–level" permission (i.e., an ACL in which the protected item is "entry"). The entry–level permissions facilitate policy expression in terms of specific Directory operations. These permissions are primarily designed to facilitate denials since denial of an entry–level permission causes the operation to fail regardless of permissions associated with attributes or Distinguished Names. Granting an operation, however, generally requires granting of both entry–level and attribute permissions (the only exception is REMOVE–ENTRY).

For the modify operations, except MODIFY–DISTINGUISHED–NAME, there is a separate entry–level permission for each operation so that denying the use of a particular modify operation is as simple as denying the entry–level permission associated with that operation. To discuss MODIFY–DISTINGUISHED–NAME, it is necessary, temporarily, to consider more than one authority. There are two cases to consider:

1. in the case where renaming does not cause the entry to "move" to a new parent in the DIT, there is a single permission category that can be used to deny the operation;

2. in the case where renaming would cause the entry to have a new parent, two entry–level permissions are involved: one is controlled by the authority for the entry with its old name, the other is controlled by the authority for the subtree area into which the renamed entry (and all its subordinates) would move.

In the first case, the authority for the entry under its old name has complete denial control and may prevent the operation by denying one entry–level permission. In the second case, both cognizant authorities would have to grant independent permissions before the operation could succeed. Either authority can deny the operation by denying one entry–level permission.

For the query operations, entry–level permissions are grouped such that READ and COMPARE have the same entry–level permission (however, they do have independent attribute permissions). Denying that permission causes both the READ and COMPARE operations to fail for all entries in the scope of the ACL with the denial.

Similarly, LIST and SEARCH have the same entry–level permission so they can only be denied as a pair.

Entry–level controls may be referred to as Operation–dependent controls.

## Type–dependent controls

The ACLs in figures 12.4 – 12.8 illustrate controls applying to a specific attribute type (e.g., type = phone number). For convenience, the standardized access control mechanisms also facilitate the expression of policy that applies to all "user attributes" present in an entry. "User attributes" include all attributes that are intended to serve the needs of the Directory user community; they do not include attributes that exist for administrative purposes (e.g., ACLs).

A feature has been provided in the access control mechanisms to allow a single ACL to apply to all user attributes in any entry that falls in the scope of the ACL. The feature allows collective control over access to the type for each user attribute in an entry or over both type and value information for all user attributes in an entry. Such controls may be referred to as *Type–dependent controls.*

## Value–dependent controls

Figures 12.4 – 12.8 also illustrate controls applying to all values present in an attribute of a particular type. This is another convenience feature of the standardized access control mechanisms; it allows easy expression of a policy that equally applies to all the values present in an attribute of a specified type. It is also possible to specify an ACL that applies to one specified value of an attribute of a specified type.

Controls expressed in terms of all the values of a particular type or in terms of a specific value of a particular type may be referred to as Value–dependent controls.

## Self–administration of group membership

Another convenience feature of the mechanisms is designed to cater to a special case of Value–dependent control. The standardized object classes for the Directory include one that is used to specify an entry that contains a group of Directory names. Such an entry could be used to hold a mailing list, an administrative grouping (see User–dependent controls), or any other useful list of Directory names.

Each entry representing a list of names has an attribute whose values are the names in the list. Controlling access to a name in a list can be achieved in the same ways used for any other attribute value. In addition, there is a feature applicable only to entries that contain lists of names. This feature facilitates the expression of policy that allows a person, whose name appears in a list, to self–administer his name with respect to that list. Self–administration might allow the person to change his name but not remove it or it might allow him to do both. It might instead allow him to remove it but not otherwise modify it. These kinds of policies are easy to express using the standardized access control mechanism.

## User–dependent controls

Figures 12.4 – 12.8 illustrate ACLs associated with policy that is applicable to all users. It is also easy to express policy that is applicable to:

1. a particular user;

2. a list of users; or

3. a collection of users whose DIT entries appear in the same (complete) subtree.

It is also easy to express policy that applies to any combination of particular users, lists of users, and users in the same subtree.

A list of users is a very flexible way of identifying a group of users for special consideration with regard to access control policy. The list can be built to include as many users as necessary; however,: it cannot contain the names of entries that represent additional lists of names. Users whose entries appear in the same subtree might all be members of the same organization or organizational unit. Policy that equally applies to an entire organization could be easily expressed using the definition of the DIT subtree representing that organization. Policy that distinguishes between organizational "insiders" (i.e., members of the organization) and "outsiders" could be easily expressed using organizational subtrees to distinguish between insiders and outsiders.

## Self–administration of user entry

The standardized mechanisms are designed to provide an easy way to express policy that allows a Directory user to self–administer the entry in the DIT which represents that user. Self–administration could be used, for example, to allow a user to modify any of the attributes in her entry, including any ACL information. Self–administration could also be restricted to allow her to to administer only a subset of the attributes in her entry.

When a user is allowed to self–administer the ACL information controlling that user's entry, there is a minor difference between the standardized access control mechanisms. Recall that there are two mechanisms being standardized in the new edition of the Directory standard. The names for the two mechanisms are *Basic Access Control (BAC)* and *Simplified Access Control (SAC)*. Self–administration of ACLs is possible under both mechanisms but it may be more convenient under BAC because it allows an ACL to be placed directly in an entry to which it applies. SAC requires each ACL to be placed outside of an entry it controls and therefore self–administration under SAC can be more complicated because the self–administered information is not collected in a single entry.

## Default controls

It is often convenient to express access control policy in terms of a general rule that applies to a wide range of users or protected items. Usually, however, there are exceptions to the rule that must be enforced. When policy is expressed in terms of a general rule with exceptions, the general rule can be thought of as a "default" control that applies unless one of the exceptions occurs.

The standardized access control mechanisms facilitate three kinds of default policies. The first is an application of Value–dependent controls, the second is an application of User–dependent controls, and the last is a very general feature whereby a precedence level is associated with each ACL. The precedence level is used to specify which ACLs prevail within a given scope of influence.

Value–dependent controls can be used to express default policy with exceptions because the access control mechanisms are designed such that an ACL that applies to a particular attribute value is considered to be "more specific" than an ACL that applies to all values of that attribute. The automated guard that makes access control decisions always favors more specific ACLs when all other decision criteria (such as precedence) are equal. Hence default policy could be expressed as an ACL with Protected Item set to "all values of attribute type $X$" while an exception could be expressed as an ACL with Protected Item set to "value $Y$ of attribute type $X$."

Similarly, User–dependent controls can be utilized to express default policy because there is also an ordering of specificity with respect to the User Class in an ACL. The ordering, from most specific to least specific, is as follows:

- User Class specified as a particular name;
- User Class specified as a list of users;

- User Class specified to include all members of a particular DIT subtree;

- User Class specified to be "all users."

Using this hierarchy of specificity, there are many strategies for expressing default ACLs and exception ACLs. An obvious strategy is to set User Class to "all users" for default controls and set it to a particular name for an exception. If the exception applies to more than one user, then the exception ACL could have User Class set to a list of users where the list contains the names to which the exception applies.

For an organization or organizational unit represented by a subtree in the DIT, the default policy for insiders could be expressed in an ACL where the User Class specifies all members of the organization's subtree; insider exceptions could be expressed in an ACL with User Class set to a list of names; outsider default policy could be expressed in an ACL where the User Class is "all users." Insider exceptions, for example, could be specified for a group of systems administrators responsible for maintaining the information in the organization's subtree. The administrators would be able to perform modify operations and read administrative attributes that are not generally available to other insiders.

In addition to specificity of User Class and Protected Item, an ACL can be assigned a precedence level that defines its relationship to other ACLs with the same scope of influence. Precedence can also be useful when ACLs have different, but overlapping, scopes of influence (ACLs defined by a single authority are allowed to freely overlap in scope of influence). When scopes overlap, access control decisions for any entry in the intersection are influenced by the precedence level of each of the ACLs involved.

The precedence level is an integer in the range from 0 to 256. A higher level takes precedence over a lower one. Precedence can be used to determine which ACL prevails when two have the same level of specificity. An important design feature of the access control mechanisms is that denials always prevail when conflicting ACLs have the same scope of influence, same specificity, and same precedence.

In making access control decisions involving specificity and precedence, the access control mechanisms consider precedence first. The mechanisms, in effect, gather all the ACLs whose:

- scope of influence includes the Protected Item for which access is requested; and

- User Class includes the user making the requested access.

Having gathered all such ACLs into a set, the mechanisms examine each ACL to determine what the set's highest precedence level is; all ACLs below that level are discarded from the set. Next, the remaining ACLs are examined to determine what the set's most specific User Class is; all ACLs with a less specific User Class are discarded from the set.

If access is being requested to an attribute value, then the set is again examined to determine what the most specific Protected Item is; all ACLs with a less specific Protected Item are discarded. The access decision is based on the ACLs remaining in the set. If there are conflicting ACLs, then denials prevail. If there are no ACLs left in the set, then the access control mechanisms automatically deny access. The set could be empty because:

there is no ACL whose scope of influence contained the requested item; or because there is no ACL whose User Class contains the requestor; or because there is no ACL for the particular Protected Item being requested.

### Hybrid Orientations

The policy orientations described above can be combined to form new, more powerful orientations that may be used when the applicability of a particular policy fragment is defined in terms of more than one of the basic orientations. For example, suppose a particular control on the READ operation applies to entries in a specific subtree except for entries in one of the branches of that subtree; further, suppose the policy only applies to a particular object class (say, object class E) within subtree levels 2 and 3. The area labeled **subtree with exclusion** in figure 12.9 is an example of the part of this hybrid orientation involving a subtree with an excluded branch.

Building on figure 12.9, an illustration of the complete scope of influence for the hypothetical policy is shown in figure 12.11 where entries with labels that begin with e (e.g., **e1**, **e2**) are the only entries of object class E. Entries labeled **e3**, **e4**, and **e6** are the only entries in the hybrid scope of influence for ACLs enforcing the example policy. Because the ACLs are expressed as hybrid subtree controls, they will automatically apply to any new entry of object class E that is added (via Add–ENTRY) to levels 2 or 3 of the area labeled **subtree with exclusion**. They will also automatically apply to any entry of object class E that is moved into the hybrid scope via the MODIFY–DISTINGUISHED–NAME operation.

Carrying the example one step further, suppose the policy applying to the hybrid scope is a modification policy controlling permissions for the ADD–ENTRY operation. Suppose the associated ACL grants entry–level permission for ADD–ENTRY to reflect a policy that the subtree with root labeled a1 may only grow by adding new entries of object class E in subtree levels 2 and 3. The ACL also grants all the needed permissions to allow addition of all applicable attribute types and values. Under this policy, an attempt to add a sibling (with object class E) of **e4** would succeed as would an attempt to add an entry of object class E as a child of **b2**. An attempt to add an entry of class E as a child of **e4** would fail but adding the same entry as a child of **e6** would succeed.

### A Preview of Multiple Authority Scenarios

In the above description of "Default controls", it was pointed out that the scope of influence for an ACL may freely overlap the scope of any other ACL. It is important to note, however, that this is true only within a part of the DIT that is controlled by a single authority. In multiple authority scenarios, the DIT is partitioned into "administrative areas" with one area (i.e., subtree) for each authority; the areas may overlap only when there is partial delegation of authority. When overlapping administrative areas have overlapping ACLs, it is possible for one or more of the overlapping ACLs to be in conflict because they do not express compatible policies; in such cases the standardized access control mechanisms provide a way to enforce

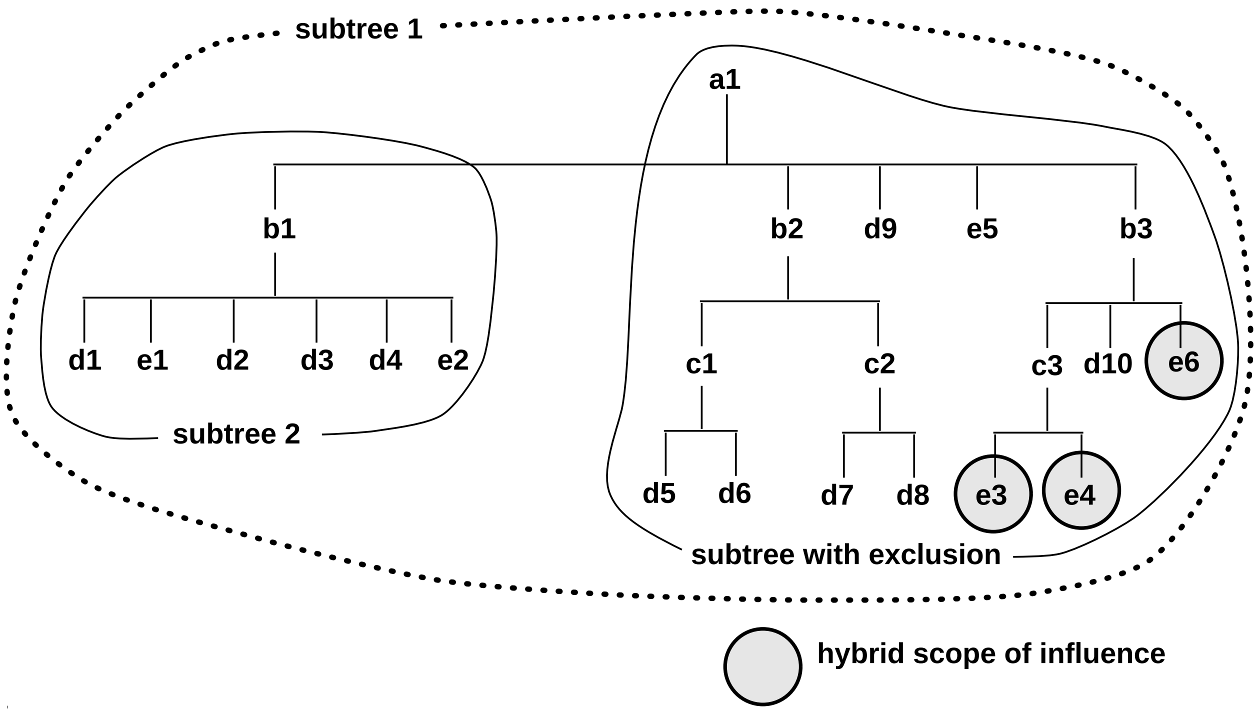



Figure 12.11: Example of hybrid scope of influence.

organizational policy regarding which authority is superior and which is subordinate. The superior authority always controls which ACL prevails.

The main point here is that the scope of influence of each ACL is limited by the boundary of the administrative area for the authority that manages that ACL. The scope of ACLs defined by a single authority may freely overlap within that authority's administrative area; they may or may not be allowed to overlap into other administrative areas, depending on the organizational relationship between the authorities. A more detailed explanation of how administrative areas limit ACL scope is provided in the section below on "Scenarios Involving Multiple Authorities."

### Use of Authentication Service by Access Control

Schemes based on access control lists are sometimes referred to as *identity–based access control* because access policy is expressed in terms of ACLs, each of which applies to one specific user or to a class of users. The automated guard uses the identity presented by the user to determine if an ACL applies to that user. One question that naturally arises is: how does the user convince the guard that the presented identity really is that user's name? A related question is: what forms of credentials will the guard accept? The Directory addresses these questions by providing a standardized *authentication service* which supports several forms of credentials. Not all forms of credentials have to be supported by all implementations of the Directory; each implementor chooses which forms its DSA product supports. A full service DSA would support three basic forms:

- name only;
- name and password;
- digital signature.

When credentials contain only a name, the guard is given very little reason to believe the claimed identity. When a password is included, the guard has more confidence in the claimed identity. The standardized access control mechanisms assume passwords are carefully administered and protected to avoid "weak" and compromised passwords. Credentials consisting of a digital signature are considered to provide the highest level of confidence in the claimed identity. The standardized access control mechanisms assume a cryptographically strong public–key method is being used; they also assume there is a "trusted" key authority that can be used when verifying a digital signature.

The access control mechanisms allow the security authority to specify, for each ACL, the kind(s) of credentials that are acceptable. An ACL that specifies a default control applying to all users (e.g., "public" users) might require name–only credentials. An ACL that grants an "insider" permission might require password credentials. An ACL that grants administrative power (e.g., granting modify permission) might require a digital signature.

Before considering an ACL that grants a permission, the guard checks to see if the requestor has satisfied the authentication requirement for that ACL. A requirement for name–only authentication, the weakest requirement, is satisfied by any of the three basic forms of credentials. A requirement for password authentication, considered a stronger requirement, may be satisfied by either a password credential or a digital signature. A requirement for digital signature authentication is the strongest requirement and is only satisfied by a digital signature. ACLs which grant a permission are ignored by the guard unless the requestor has satisfied the authentication requirement specified in that ACL.

The authentication requirement in an ACL that denies a permission indicates the minimum level the requestor must satisfy in order not to be denied access. An ACL that denies and requires authentication via digital signature will, in effect, deny the permission to all users that do not authenticate with a digital signature. This is true regardless of the User Class specified in the ACL — a user who does not authenticate with a digital signature cannot adequately convince the guard that the denial does not apply. For users that do authenticate with a digital signature, the User Class in the ACL will determine whether or not the denial applies.

An interesting wrinkle in standardized authentication occurs when more than one DSA is involved in servicing a request. A request may be initially submitted to a DSA that does not contain the requested information. In such a case, there are situations in which the DSA automatically contacts another DSA and passes on the request; propagation of the request from one DSA to the next might occur several times before a DSA containing the requested information is found. When a request is passed among DSAs, it is referred to as a *distributed operation*.

There is one distributed operation scenario in which the final DSA is able to perform authentication of the user. This would be the case only if the request had been digitally signed by the user. The final DSA would then be able to verify that signature and apply the results to access control decision making.

In all other distributed operation scenarios, the final DSA cannot directly authenticate the user. For example, when password authentication is used, the initial DSA is the only one that receives the password and there is no standardized way for it to propagate the password to other DSAs. Also, it is possible that the user was authenticated only once by digitally signing a request to begin the Directory session. When authentication occurs only once at the beginning of a session, there is no way for other DSAs to receive the digital signature. Another new feature of the new edition of the Directory standard does, however, provide a way for the initial DSA to optionally indicate what authentication it performed. The indication is included in the operation arguments passed from one DSA to another.

The new feature allows the final DSA to make access control decisions based on user authentication performed by the initial DSA. This can only occur, however, when the final DSA trusts the first DSA and all intermediate DSAs to have properly handled the indicator of authentication level. Before the final DSA receives the indicator, any of the propagating DSAs could have purposefully caused it to be incorrect. This brings up questions about how a DSA knows what other DSAs it "trusts" and what happens when one of the propagating DSAs is not trusted.

The first question is addressed in implementation agreements produced by the OSE Implementors' Workshop. Under those agreements, each DSA maintains an internal list of the names of other DSAs it trusts to handle authentication processing properly ("proper handling" is defined locally by each security administrator). A security administrator might also want to require each DSA to digitally sign the request so that there is high confidence in the identity of each propagating DSA.

When one of the propagating DSAs is not trusted, or when the authentication indicator is missing from a propagated request, the final DSA may assume no authentication was performed (i.e., name–only credentials), or it can make use of yet another new feature of the new Directory that allows the DSA to respond to the request by returning a "referral" to itself. Ordinarily, referrals are used to indicate other DSAs which may be able to respond to a request. However, when a DSA issues a referral to itself, it is, in effect, requesting the user to resubmit the request directly to that DSA so there are no propagating DSAs to trust. If the request is resubmitted directly, the DSA that issued the referral to itself is able to directly authenticate the user and use the results in making access control decisions.

### 12.2.2 Scenarios Involving Multiple Authorities

The Directory administrative model defines three aspects of administrative authority: access control, schema, and collective attributes (i.e., attributes that are common to several entries).

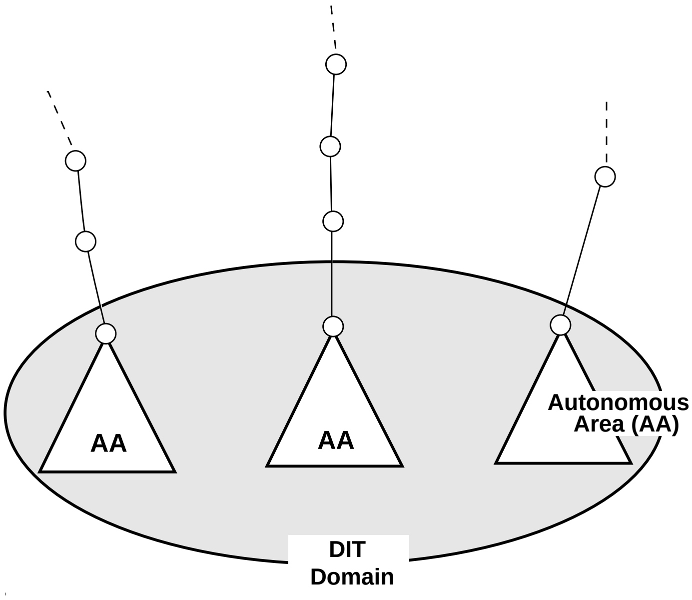



Figure 12.12: Example DIT Domain.

This section discusses how authority is modeled for access control and also points out how access control authority and schema authority are different. Collective attributes are not outside the scope of this study.

### Multiple Security Authorities

The preceding discussion has primarily focused on a rather contrived situation in which there is only one security authority controlling the entire DIB. In this section we consider a more realistic world in which there are many security authorities, each exercising control over a well–defined part of the DIT. The single authority scenario is still entirely valid for parts of the DIT that are autonomously controlled by one authority who chooses not to delegate any part of that authority. In the more complicated world of multiple authorities, there are many autonomous authorities and there are various forms of delegation from autonomous authorities to subordinate authorities. This section describes administrative concepts used to accommodate multiple autonomous security authorities and delegation. Related parts of the standardized access control mechanisms are also discussed. Administration of the Directory is based on the central concept of a *DIT Domain.* A DIT Domain consists of one or more independent (i.e., non–overlapping) subtrees of the DIT which are under the control of a single organization. Each of the independent subtrees in a domain is referred to as an Autonomous Administrative Area or, more simply, an *Autonomous Area (AA).* Figure 12.12 illustrates a DIT Domain consisting of three AAs. Each of the triangles in figure 12.12 represents a complete subtree.

Access control decisions regarding the contents of an AA are based solely on ACLs ap-

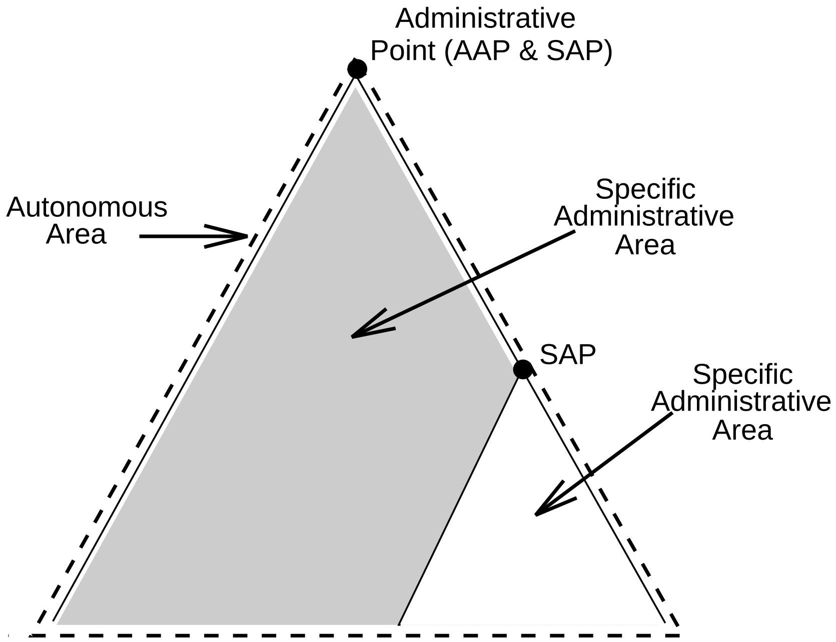



Figure 12.13: Example of full delegation of authority.

pearing in that AA. An ACL that applies to more than one AA must be repeated in each of those AAs. The organization controlling a particular DIT Domain may choose to have separate security authorities for each AA, or more than one AA could be assigned to a single security authority. It is a matter for security policy to specify the relationship between AAs and security authorities.

Each AA can further be divided into sub–areas to accommodate delegation of authority. Two basic forms of delegation are supported: full delegation and partial delegation. Each instance of delegation must be defined in terms of a complete subtree of the AA for which the delegation applies. Figure 12.13 illustrates an AA which has been divided into two non–overlapping sub–areas to accommodate a full delegation of authority from the authority controlling the AA to a subordinate authority. In this case, each sub–area is called a *Specific Administrative Area (SAA)*. Access control decisions regarding the contents of each SAA are based solely on ACLs appearing in that SAA. Even if an ACL was incorrectly given a scope of influence that exceeds the boundary of the SAA in which it is defined, the access control mechanisms would automatically ignore the invalid part of the ACL scope. When making an access decision, the automated guard first identifies which SAA the requested item is in and then only considers ACLs defined within that SAA, so any ACL from another SAA (regardless of its scope of influence) will be ignored.

Figure 12.13 also uses some basic terminology for the root entry of an administrative area. The root entry for an AA is called an *Autonomous Administrative Point (AAP)* while the root of an SAA is called a *Specific Administrative Point (SAP)*.

Figure 12.14 illustrates an instance of partial delegation of authority. The partially

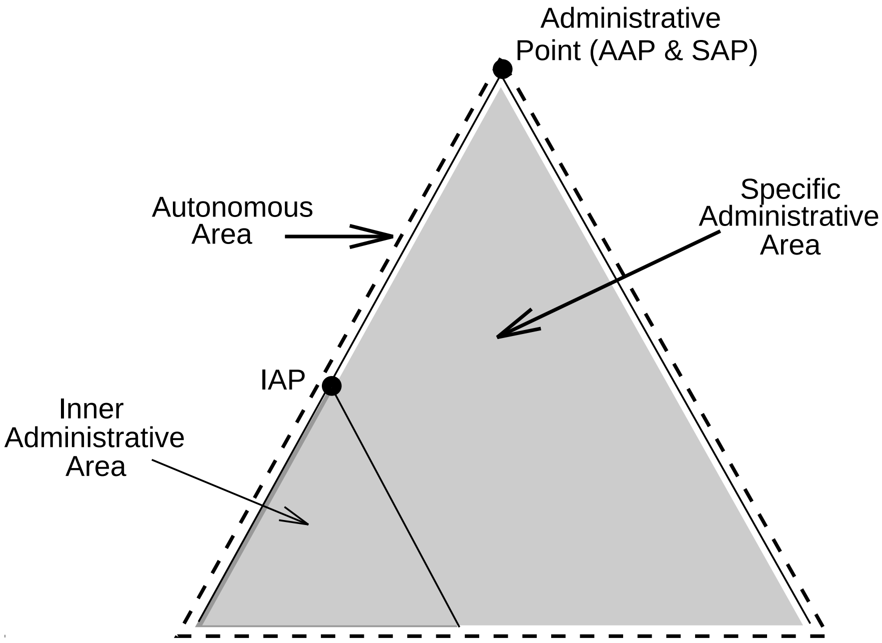



Figure 12.14: Example of partial delegation of authority.

delegated area is called an *Inner Administrative Area (IAA)*, its root entry is called an *Inner Administrative Point (IAP)*. Figure 12.14 shows the two administrative areas overlapping to indicate that the authority for the SAA may define ACLs that affect decisions about the contents of the IAA. IAAs are the only type of administrative area allowed to overlap another administrative area. Additional nested IAAs could be defined inside the IAA in figure 12.14 to reflect lower levels of partial delegation. Security policy should address the need for IAAs and nested IAAs. Security policy should also specify that the authority controlling an IAA shall not create an SAP within the IAA — this precludes the possibility of the delegated authority being able to contravene policy set by the superior authority. The superior authority may wish to enforce this policy fragment by specifying an ACL in the SAA such that the ACL's scope of influence includes the entire IAA and it denies ADD of a particular attribute value needed to build a new SAP.

When a superior authority partially delegates to a subordinate authority, the superior is able to retain control over any aspect of ACL policy because:

- when the requested item is inside an IAA, the guard takes into consideration all ACLs defined for that IAA, ACLs defined for any enclosing IAA, and ACLs defined in the enclosing SAA; and

- ACLs defined in the enclosing SAA are allowed to have a scope of influence that reaches into any enclosed IAA; and

- the authority for the SAA can define a maximum precedence value for each delegated authority such that ACLs defined for the SAA can be given a precedence that is guaranteed to be higher than any assigned by a delegated authority.

Security policy defines the maximum precedence that may be used by each delegated authority. The authority for the SAA may then override any ACL defined by a subordinate authority by simply using a precedence that is higher than the maximum assigned to that subordinate. The assigning and enforcement of maximum precedence is outside the scope of the Directory standard; it is anticipated, however, that implementors will provide a way for the superior authority to specify and enforce the maximum precedence assigned to each subordinate authority.

A superior authority may also define ACLs with a precedence that is much lower than the maximum assigned to a subordinate. In this case, the superior is defining, in effect, default controls which may be overridden by the subordinate by using a higher precedence value.

Having discussed what full and partial delegation are, in the context of the standardized access control mechanisms, we can now emphasize the basic difference between the two mechanisms. One of the mechanisms, known as Basic Access Control (BAC), supports the full range of delegation relationships discussed above. The other mechanism, known as Simplified Access Control (SAC), supports full delegation only.

### Relationship Between Security Authority and Schema Authority

This chapter has so far taken a narrow view of Directory management and policy by focusing on security policy aspects of Directory management. Another important part of Directory management is concerned with controlling aspects of DIT structure and content such as:

- rules defining the allowable DIT parent – child relationships in terms of object classes (e.g., a policy might specify that a parent entry of object class organization may only have children entries of object class person); and
- rules defining the mandatory and optional attributes for each object class.

Definition and enforcement of such rules is known as *Schema Management.* The general administrative model for the Directory defines schema authority and allows that authority to be separate from the security authority for a given autonomous area.

Security management using BAC or SAC is sometimes confused with schema management since the standardized access control mechanisms could, conceivably, be used to enforce certain simple DIT structure rules. For example, a hybrid of Class-dependent and Level-dependent controls could be used to enforce, in a limited way, the allowable parent – child object class relationships in the DIT.

There is no way, however, for an ACL to fully express a schema policy like: "Entries of object class $X$ shall contain attributes $A$, $B$, and $C$; further, entries of object class $X$ may optionally contain attributes E and F (attributes other than $A$, $B$, $C$, $E$, and $F$ are not allowed in an entry of object class $X$)."

ACLs could prevent an ADD or MODIFY–ENTRY from placing an attribute other than $A$, $B$, $C$, $E$, or $F$ in an entry of object class $X$; but ACLs cannot enforce policy on mandatory attributes for a given object class. Mandatory attributes are attributes that must appear in each instance of the specified object class. ACLs can only keep attributes out of an entry, they cannot force certain attributes to be in an entry.

Because the schema authority may be different from the security authority, and because many schema policies cannot be expressed in ACLs, the mechanism that enforces schema policy is assumed to be completely independent of the mechanism used to enforce automated access control policy. Note that conflicts can arise between access control policy and schema management policy. For example, schema policy may specify that an entry of object class X shall contain attribute type A while access control policy denies the ability to add attribute type $A$ to an entry of object class $X$. There is a need, therefore, for coordination between access control authorities and schema authorities.

A final point concerning schema rules is that they can be used to support management auditing procedures that may be part of monitoring the activity of security authorities. Specifically, schema rules can be expressed which require each entry to contain the following attribute types:

- createTimestamp: indicates the time of creation of an entry;
- modifyTimestamp: indicates when an entry was last modified;
- creatorsName: indicates the name of the user performing the ADD–ENTRY operation;
- modifiersName: indicates the name of the user performing the MODIFY–ENTRY operation.

These attributes are standardized in the new edition of the Directory standard. They are defined such that each is completely under the control of the DSA; no user modification is allowed. Theoretically, this means security authorities could not change their values. Also, note that the modifyTimestamp and modifiersName attributes do not really provide an audit trail since each modify operation causes the previous values of these attributes to be overwritten. They can only be used to monitor when the most recent modify was applied and who did it.

### 12.2.3 The Hazards of Data Caching

As previously mentioned, a DSA may, under certain circumstances, pass an operation request on to other DSAs until a DSA is found which contains the DIB information needed to respond to the request. The response is then "chained" back through each of the DSAs that propagated the request. There is, therefore, always the possibility that a propagating DSA may copy the request or the response, or both. Subsequently, that DSA could disclose information from the result without enforcing ACLs defined in the DSA that generated the result. Allowing results to be passed back through a DSA chain may, therefore, result in violations of an organization's security policy.

This does not mean, however, that all replication of DIB data must be banned; such a ban is probably impossible to enforce and would cause the loss of advantages afforded by replicated data in a distributed database environment (advantages include increased availability of data and reduced response times). The new edition of the Directory standard contains another new feature called *shadowing* which provides a disciplined way to replicate data such that security policy is not violated. When information is shadowed, the standard specifies three important requirements:

1. each unit of replication shall include all relevant ACLs; and

2. each DSA using shadowed information shall enforce relevant ACLs exactly as they are enforced in the DSA that provided the shadowed information; and

3. shadowed information shall not be modified (only the master copy of each entry may be modified).

A shadowing agreement also addresses how often the shadow is refreshed and which DSA is responsible for providing refreshed data.

However, even in implementations of the new standard, DSAs can still choose to copy distributed operation results and thereby gain copies of DIB information which do not include the relevant ACLs. This form of undisciplined replication is referred to as *results caching.* The potential problem for security authorities is that there is no effective way for the standard to preclude it.

Security policy should, therefore, address the problem of caching and provide policy guidelines about whether or not it is deemed a serious threat. In cases where it is considered to be a serious threat, Security policy can specify that measures are to be taken to avoid caching. Such measures can include requirements such as:

- results may only be passed back through a chain of "trusted" DSAs (each DSA contains an internal list of trusted DSAs as previously discussed);

- when a chain cannot be trusted, the DSA may refuse service or return a referral to itself (as discussed in the section on "Use of Authentication Service").

### 12.2.4 Policy Aspects That Are Not Supported

Recall that the new Directory access control mechanisms do not support certain high–level policy orientations such as capabilities–based access control. This section lists some additional aspects of security policy that cannot be directly approached using the standardized mechanisms.

1. The standardized ACLs do not allow access permissions to be directly dependent on the time of day or date of the request. Time–dependent controls could be effected indirectly by using a default control which is periodically overridden by adding a higher precedence ACL. The override ACL would have to be manually removed at the point in time when the default control is to resume (actually it would not have to be completely removed, the precedence level could be lowered to eliminate its effect).

2. The standardized ACLs do not allow access permissions to be dependent on the point of origin of the request.

3. The standardized access control mechanisms do not support access control policies that make access decisions dependent on what has happened in the past.

4. The standardized access control mechanisms do not support policy involving requirements for encryption to achieve secrecy during computer interactions.

5. The standardized access control mechanisms do not directly control the depth of a subtree that may be accessed during a SEARCH operation. Level–dependent controls can be used to preclude the use of a particular level of a subtree by any SEARCH operation, but this does not flexibly support general policy statements such as: "SEARCH operation results shall not return more than 3 levels of subtree information."

6. The standardized access control mechanisms do not support access control policies regarding information disclosed in a continuation reference; more generally, the mechanisms do not address control of information known as knowledge which is used to allow a DSA to know that other DSAs exist and which objects the other DSAs have directly available. Continuation references occur in a referral and may also form part of a SEARCH result.

# Bibliography


[AMPH87] Abrams, Marshall, Podell, and Harold. *Computer and Network Security.* Catalog No. EH0255-0. IEEE Computer Society Press, 1987.

[Ank92] R. Ankney. Security Services in Message Handling Environments. *The Messaging Technology Report*, 1(5), June 1992.

[ANS85] Financial Institution Key Management (Wholesale) Standard. American National Standard X9.17, American National Standards Institute, 1985.

[ANS86] Financial Institution Message Authentication (Wholesale). Technical Report X9.9, American National Standards Institute, 1986.

[ANS89] Database Language - SQL with Integrity Inhancements. American National Standard X3.135, American National Standards Institute, 1989.

[ANS92] Database Language SQL. American National Standard X3.135-1992, American National Standards Institute, 1992.

[ATT90] AT&T. *UNIX System V Release 4 Network User's and Administrator's Guide*, 1990.

[Bel89] Steven M. Bellovin. Security Problems in the TCP/IP Protocol Suite. *Computer Communications Review*, 9(2):32–48, April 1989.

[bel90] Integrated Information Systems Architecture Seminar. Bell Atlantic, February 22 1990.

[Bel92] Steven M. Bellovin. There Be Dragons. In *USENIX Security Symposium III Proceedings*, pages 1–16. USENIX Association, September 14-16 1992.

[CA-92] CERT Advisory: Altered System Binaries Incident. CERT, June 22 1992.

[CA-93] CERT Advisory: Anonymous FTP Activity. CERT, July 14 1993.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security.* Addison-Wesley, Reading, MA, 1994.

[CCI88a] X.400, Message handling system and service overview. CCITT, 1988.

[CCI88b] X.402, Message handling systems: Overall architecture. CCITT, 1988.

[CCI88c] X.411, Message handling systems - Message transfer system: Abstract service definition and procedures. CCITT, 1988.

[CCI88d] X.509, The Directory - Authentication Framework. CCITT, 1988.

[Cha92] D. Brent Chapman. Network (In)Security Through IP Packet Filtering. In *USENIX Security Symposium III Proceedings*, pages 63–76. USENIX Association, September 14-16 1992.

[Che90] William R. Cheswick. The Design of a Secure Internet Gateway. In *USENIX Summer Conference Proceedings*. USENIX Association, June 1990.

[CM89] D. Rush C. Mitchell, M. Walker. CCITT/ISO Standards for Secure Message Handling. *IEEE Journal on Selected Areas in Communications*, 7(4), May 1989.

[Com86] Federal Communications Commission. *Computer Inquiry III*. FCC, June 1986.

[Cou89] National Research Council. *Growing Vulnerability of the Public Switched Networks*. National Academy Press, 1989.

[Cou90] National Security Telecommunications Advisory Council. *Report of the Network Security Task Force*. National Security Telecommunications Advisory Council, 1990.

[CTC93] The Canadian Trusted Computer Product Evaluation Criteria (CTCPEC) Version 3.0e. Canadian System Security Centre, Communications Security Establishment, Government of Canada, January 1993.

[Cur92] D. Curry. *UNIX System Security*. Addison-Wesley Publishing Company, Inc., 1992.

[Dol88] S.E. Dolan. Open Network Architecture from an Operational Perspective. In *IEEE Globecom*. IEEE, 1988.

[Dwo91] F.S. Dworak. Approaches to Detecting and Resolving Feature Interactions. In *Proceedings, IEEE Globecom*. IEEE, 1991.

[Fah92] Paul Fahn. Answers to Frequently Asked Questions About Today's Cryptography. RSA Laboratories, 1992.

[FC92] Federal Criteria for Information Technology Security Version 1.0. National Institute of Standards and Technology and National Security Agency, December 1992.

[FIP85] Computer Data Authentication. Federal Information Processing Standards Publication FIPS 113, National Bureau of Standards (U.S.), May 30 1985.

[FIP90] Database Language SQL. Federal Information Processing Standard 127-1, National Institute of Standards and Technology, 1990.

[FIP92] Key Management Using ANSI X9.17. Federal Information Processing Standards Publication 171, National Institute of Standards and Technology, April 27 1992.

[FIP93a] Database Language SQL. Federal Information Processing Standard 127-2, National Institute of Standards and Technology, January 1993.

[FIP93b] Portable Operating System Interface (POSIX) - System Application Program Interface [C Language]. Federal Information Processing Standard 151-2, National Institute of Standards and Technology, May 12 1993.

[FIP93c] Draft Digital Signature Standard (DSS). Federal Information Processing Standard, National Institute of Standards and Technology, February 1 1993.

[FIP93d] Draft Standard Security Label for Information Transfer. Federal Information Processing Standard, National Institute of Standards and Technology, September 30 1993.

[FIP94] Security Requirements for Cryptographic Modules. Federal Information Processing Standards Publication 140-1, National Institute of Standards and Technology, January 11 1994.

[Fis93] G. Fisher. *Application Portability Profile (APP) The U.S. Government's Open System Environment Profile OSE/1 Version 2.0.* NIST Special Publication 500-187. National Institute of Standards and Technology, June 1993.

[For94] Warwick Ford. *Computer Communications Security.* Prentice-Hall, Englewood Cliffs, NJ, 1994.

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

[Hel90] Dan Heller. *XView Programming Manual.* O'Reilly & Associates, Inc., 1990.

[iee86] Helping Computers Communicate. *IEEE Spectrum*, March 1986.

[ISO90a] Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application Program Interface (API) [C Language]. ISO/IEC 9945-1, 1990.

[ISO90b] Remote Database Access - Part 1: Generic Model. ISO/JTC1/SC21 N4282, Information Processing Systems - Open Systems Interconnect, 1990.

[ISO90c] Remote Database Access - Part 2: SQL Specialization. ISO/JTC1/SC21 N4281, Information Processing Systems - Open Systems Interconnect, 1990.

[ISO92] ISO-ANSI Working Draft Database Language SQL (SQL3). ISO/IEC JTC1/SC21 N6931, ISO/IEC, July 1992.

[ITS91] Information Technology Security Evaluation Criteria (ITSEC). Department of Trade and Industry, London, June 1991. Harmonized Criteria of France, Germany, the Netherlands, and the United Kingdom;.

[JS92] Saqib Jang and Vipin Samar. Network Information Service Plus (NIS+): An Enterprise Naming Service. Solaris 2.0 White Papers, SunSoft, 1992.

[Klu92] H.M. Kluepfel. A Systems Engineering Approach to Security Baselines for SS7. Technical Report TM-STS-020882, Bellcore, 1992.

[Koh91] J.T. Kohl. The Evolution of the Kerberos Authentication Service. In *Proceedings - Spring 1991 EurOpen Conference*, 1991.

[Koh92] J.T. Kohl. The Kerberos Network Authentication Service (V5), RFC, Revison #5. MIT, April 1992.

[Kuh91] D.R. Kuhn. IEEE's POSIX: Making Progress. *IEEE Spectrum*, December 1991.

[LeF92] William LeFebvre. Restricting Network Access to System Daemons Under SunOS. In *USENIX Security Symposium III Proceedings*, pages 93–104. USENIX Association, September 14-16 1992.

[Lin90] J. Linn. Practical Authentication for Distributed Computing. In *1990 Security and Privacy Symposium.* IEEE CS Press, May 1990.

[Nec92] James Nechvatal. A Public-Key Certificate Management System. National Institute of Standards and Technology, May 1992.

[NIS91a] Advanced Athentication Technology. NIST Computer Systems Laboratory Bulletin, NIST, November 1991.

[NIS91b] *Public-Key Cryptography.* NIST Special Publication 800-2. National Institute of Standards and Technology, April 1991.

[OB91] Karen Olsen and John Barkley. *Issues in Transparent File Access.* NIST Special Publication 500-186. National Institute of Standards and Technology, April 1991.

[PI93] W. Timothy Polk and Lawrence E. Bassham III. *Security Issues in the Database Language SQL.* NIST Special Publication, 800-8. National Institute of Standards and Technology, August 1993.

[POS92a] Draft Guide to the POSIX Open Systems Environment. P1003.0/D16, IEEE, 1992.

[POS92b] Draft Standard for Information Technology - Portable Operating System Interface (POSIX) - Amendment: Protection, Audit and Control Interfaces. P1003.1e/D13, IEEE, November 1992.

[POS92c] Draft Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 2: Shell and Utilities - Amendment: Protection and Control Utilities. P1003.2c/D13, IEEE, November 1992.

[POS93] Draft Standard for Transparent File Access Amendment to Portable Operating System Interface (POSIX). P1003.1f/D8, IEEE, November 1993.

[PR91] Holbrook P. and J. Reynolds. RFC 1244: Security Policy Handbook. prepared for the Internet Engineering Task Force, 1991.

[Ran92] Marcus Ranum. An Internet Firewall. In *World Conference on Systems Management and Security*, 1992.

[Ran93] Marcus Ranum. Thinking About Firewalls. In *SANS-II Conference*, April 1993.

[Rap93] Raptor Systems Incorporated. *Eagle Network Security Management System, User's Guide*, 1993.

[Ros90] Marshall T. Rose. *The Open Book.* Prentice-Hall, 1990.

[Ros91] David S. H. Rosenthal. *Inter-Client Communication Conventions Manual.* MIT X Consortium, 1991. MIT X Consortium Standard. X Version 11, Release 5.

[RS89] J. Gettys R.W. Scheifler. The X Window System. *ACM Transactions on Graphics*, 5(2), 1989.

[Rus91] G. T. Russell, Deborah & Gangemi Sr. *Computer Security Basics.* O'Reilly & Associates, Inc., 1991.

[Sch91] Robert Scheifler. *X Security.* MIT X Consortium, 1991. MIT X Consortium Standard. X Version 11, Release 5.

[SH88] G. Giridharagopal S. Homayoon. ONA: Demands on Provisioning and Performance. In *IEEE Globecom.* IEEE, 1988.

[Sim88] L. Simpson. Open Network Architecture: OAM Perspective, an RBOC's View. In *IEEE Globecom.* IEEE, 1988.

[SMS87] J.I. Schiller S.P. Miller, B.C. Neuman and J.H. Saltzer. Kerberos Authentication and Authorization System. Section E.2.1, MIT Project Athena, December 21 1987.

[SQ92] Carl-Mitchell S. and John S. Quarterman. Building Internet Firewalls. *Unix-World*, pages 93–102, February 1992.

[SUN90a] Sun Microsystems Inc. *Network Programming Guide*, Revision A March 27 1990.

[SUN90b] Sun Microsystems Inc. *System & Network Administration*, Revision A March 27 1990.

[TA91] J.J. Tardo and K. Alagappan. SPX: Global Authentication Using Public Key Certificates. In *Proc. IEEE Symp. Research in Security and Privacy*. IEEE CS Press, 1991.

[TCS85] Trusted Computer System Evaluation Criteria. DOD 5200.28-STD, National Computer Security Center, December 1985.

[TDI91] Trusted Database Management System Interpretation. NCSC-TG 021, National Computer Security Center, April 1991.

[TNI90] Trusted Network Interpretation. NCSC-TG 005, National Computer Security Center, August 1990.

[Ven92] Wietse Venema. TCP Wrapper: Network Montoring, Access Control and Booby Traps. In *USENIX Security Symposium III Proceedings*, pages 85–92. USENIX Association, September 14-16 1992.

[WL92] T.Y.C. Woo and Simon S. Lam. *Authentication for Distributed Systems*. IEEE CS Press, 1992.

[Woo87] J. P. L Woodward. Security Requirements for System High and Compartmented Mode Workstations. Technical Report MTR 9992, Revision 1, The MITRE Corporation, Bedford, MA, November 1987. Also published by the Defense Intelligence Agency as document DDS-2600-5502-87.

# Appendix A

# ISO Protocol Security Standardization Projects

**Anastase Nakassis**

This Appendix contains background and status information on the broad spectrum of ongoing and emerging standards projects related to ISO protocols. The material contained in this Appendix is necessarily subjective representing the opinion of the author on the state of affairs in several standards projects, the results of which have yet to be readily available to users. Such information is intended to provide a perspective on the directions being taken to provide security in an ISO network environment. For more information on ISO Standards activities related to communications security see [For94].

## A.1 Introduction

Recently, a great amount of effort has been expended towards the development of generic security standards. Indeed, while several standards (such as mail, directory, and file transfer) had incorporated security within the protocol, it was felt that these solutions were ad-hoc, weak, and the wrong thing to do. Indeed, a proliferation of application specific security mechanisms would be bound to result in systems that would be hard to manage and whose security profile would be impossible to assess.

The work towards more generic solutions appears to be two pronged:

- Work on lower layers security has been initiated and rapidly progressed within (layer two) IEEE and SC6 (layers three and four). This work is quite mature.
- Work on upper layer security is pursued within ISO (SC21 and SC27), CCITT, and ECMA, sometimes on an ad-hoc basis. While ECMA has passed several standards and the work on directories is quite mature, most of the upper layer standards are either immature or generic blueprints.

Thus, at this time it would appear that ISO communication standards supporting communication integrity and confidentiality are around the corner, but that the wait for upper layer solutions will be substantially longer.

The paragraphs that follow will attempt to present the ongoing security activities of which the author is aware, be it through direct participation or by document scanning. Given the fluidity of this area and the fact that documents incorporate a built-in lag, this compendium should not be expected to be accurate and up-to-date in all of its particulars. In addition, all included judgments reflect the author's opinion and are not necessarily the consensus of those active in security standards.

## A.2 Acronyms and Terminology

While the author does have the intention to use as few acronyms as possible, he suspects that he may at times fall victim to acronym temptation. Therefore, this section will serve as a repository of the different acronyms used through the text.

**ACSE** Association Control Service Element; an Application Service Element that manages associations.

**ANSI** American National Standards Institute.

**CCITT** Comité Consultatif International Télégraphique et Téléphonique which loosely translates into International Consultative Committee for Telegraphy and Telephony.

**CD** Committee Draft. A Committee Draft is the next step (in ISO) from a Working Draft. Used to be known as a Draft Proposal (DP).

**CLNP** Connectionless Network Protocol.

**DEA** Data Encryption Algorithm.

**DIS** Draft International Standard. The next step for a CD (or DP) on its way to becoming an ISO IS.

**DP** Draft Proposal. This term is currently obsolete and the acronym CD (Committee Draft) is used instead.

**ECMA** European Computer Manufacturers Association.

**EDI** Electronic Data Interchange.

**EESP** End System to End System Security Protocol.

**ESO** External Security Object.

**ETSI** European Telecommunications Standardization Institute.

**EWOS** European Workshop for Open Systems.

**FTAM** File Transfer Access Method.

**GSS-API** Generic Security Service Application Program Interface.

**GULS** Generic Upper Layer Security service element.

**IEC** International Electrotechnical Commission.

**IEEE** Institute for Electrical and Electronics Engineers.

**IETF** Internet Enginnering Task Force.

**IS** International Standard. The final incarnation of an ISO document.

**ISO** International Organization for Standardization.

**JTC1** Joint (ISO and IEC) Technical committee 1.

**MHS** Message Handling Systems. The title of a joint (CCITT-ISO) multipart standard known as the X.400-X.420 CCITT Recommendations or as ISO 10021, parts 1-7. The ISO equivalent of MHS is Message Oriented Text Interchange System.

**MOTIS** See MHS above.

**NLSP** Network Layer Security Protocol; the OSI protocol that subsumes SP3.

**NWI** New Work Item. Before an ISO committee officially starts work towards a standard, it needs to propose (and be granted approval of) a new work item.

**ODA** Office Document Architecture.

**OSI** Open Systems Interconnection.

**PDAD** Proposed Draft Addendum.

**PDU** Protocol Data Unit.

**PIN** Person Identification Number.

**Q** Question, the CCITT equivalent of NWI.

**SC** Subcommittee.

**SG** Study Group; CCITT is organized into 15 Study Groups.

**SDNS** Secure Data Network System.

**SILS** Standards for Interoperable Local Network Security.

**SP3** Security Protocol for layer three; a protocol in the SDNS series.

**SP4** Security Protocol for layer four; a protocol in the SDNS series.

**TLSP** Transport Layer Security Protocol.

**ULSM** OSI Upper Layer Security Model.

**WD** Working Draft. The earliest and most immature instance of a document in the ISO standardization process.

**WG** Working Group.

## A.3 ISO Existing and Nascent Standards

### A.3.1 Introduction

### A.3.2 Security work within SC6

SC6 is currently working on three major security standards:

- NLSP
- TLSP
- OSI Lower Layer Security Model

#### TLSP

TLSP (Transport Layer Security Protocol) is the linear descendant of SP4 of the SDNS series. It is assumed to run at the bottom of the transport layer and to provide security services, whenever such services are needed, on a per-connection basis.

In essence, TLSP provides cryptographic transformations which are end-to-end and provided directly above the network layer.

TLSP is an IS as of July 1992.

#### NLSP

NLSP, like TLSP, is the direct descendant of the corresponding SDNS document (SP3). But, unlike TLSP, NLSP has evolved into a much more complicated protocol that incorporates facilities for key management and synchronization between NLSP peers.

At its inception, NLSP was supposed to be at the top of layer 3 and to provide support for a functionality virtually identical to the TLSP functionality. Initially, this caused several parties (such as the UK) to ask that a single Lower Layer be developed and be placed between layers three and four. Nevertheless, since NLSP was supposed to run in conjunction with X.25 (which necessitated a different NLSP placement) this approach was abandoned.

At this point NLSP is quite a complicated protocol. To start, those in favor of a single security protocol must accept the fact that the connectionless NLSP and the connection-oriented NLSP are different protocols. In addition, NLSP includes multiple functional areas, not the least of which is key management. Key management forces NLSP to reinvent transport-like mechanisms within layer three.

NLSP supports cryptographic protection either between End Systems (and in this case resembles TLSP) or between Intermediate Systems that are located at the borders of security domains. This latter aspect makes NLSP quite appealing to those who would like to provide security services not by securing each and every system in a domain but by forcing all external communications to transit through a small set of secure systems (assuming that communications within the domain need no security services). In this sense, one can see NLSP as supporting (at the domain level) administrative policies (mandatory security) while TLSP is more tuned towards discretionary communication policies. The problem nevertheless is that the requirement that NLSP be deployable in Intermediate Systems (ISs) causes considerable complications which cannot be addressed seamlessly and without considerable architectural constraints.

NLSP advanced to DIS in July of 1992. Nevertheless, it seems that there may be additional difficulties ahead for the following reasons:

- NLSP is a fairly complex protocol;

- Work on an NLSP look-alike has been initiated in IETF. The experts who reviewed NLSP for possible adoption were less than enthusiastic. Thus, an NLSP competitor may emerge within IETF.

- The OSI model appears to consist of a set of protocol stacks. Within each stack, the protocols used are entered in a fixed succession. Nevertheless the complexities of NLSP within ISs results in a complex model which appears suboptimal and at odds with OSI practice up to now.

  Indeed, there may be recursive calls to NLSP and the forwarding protocol (e.g., CLNP) in order to handle alternating sequences of encryption and fragmentation.

- Criticisms are mounting on the aspects of NLSP that pertain to the connection oriented forwarding.

Thus, it appears that NLSP will be challenged before it advances to IS.

### A.3.3 Lower Layer Security Model

In its first draft, this document attempted to chisel in stone specific architectural preferences. This caused substantial conflicts and for some period the document became simply a set of Guidelines. A revival has since occurred. In its present form, the model addresses the following issues:

- The concepts that are generally applicable to Lower Layer Security Standards;
- General Guidelines for the selection and placement of security services and mechanisms;
- interactions between the layers (when at least one is a lower one) relating to security; and
- general requirements for security management across the lower layers.

### A.3.4 Security work within SC21

The security work within SC21 falls roughly within three categories:

- A set of security frameworks which seek to expand 7498-2 (the OSI security architecture) and to provide more detailed information.
- Work geared towards providing generic security tools and solutions.
- Work that is geared towards supporting security for specific applications such as FTAM, mail, and directories.

This subsection will provide a quick overview of these areas.

#### Security frameworks

The basis of all security work within ISO is ISO 7498-2, the OSI Security Architecture. This standard provides text and definitions that cover the following:

1. security attacks relevant to Open System,
2. general architectural elements that can be used to thwart such attacks, and
3. circumstances under which the security elements can be used.

Such a document is, by its very nature, broad in scope and covers principles rather than detailed solutions. It leaves a wide latitude as to which elements can be used and where specific threats can be met.

SC21/WG1 is currently developing a multipart standard which consists of Security Framework documents. Each part aims to provide comprehensive and consistent coverage of each specific security functional area and to define the range of mechanisms that can be used to support each security service. The following Frameworks are developed within WG1:

1. **Framework Overview**

   This document provides the glue that binds the other frameworks together. That is:

- it defines the Security concepts that are required in more than one framework standard, and
- describes the inter-relationships among the services and mechanisms identified in other frameworks.

This document is currently a Committee Draft.

2. **Guide to Open System Security**

   This document provides an overview of all known and relevant Security activities. It is a document similar in scope to this report and is one of the report's primary sources.

   Currently, this document is a Working Draft and is maintained as a living document.

3. **Authentication Framework**

   This framework was the first framework to be advanced to CD status (August 1990) and was quickly progressed to DIS. But, it has since stalled, its editor has resigned, and the timetable for its progression to IS is clouded in doubt.

   This document describes all aspects of Authentication (e.g., a remote logon) as these apply to Open Systems. In particular,

   - it defines the basic concepts of Authentication;
   - it classifies the authentication mechanisms;
   - it defines the service each class provides;
   - it identifies the functional requirements for protocols to support these classes of mechanisms; and
   - it identifies the management requirements for supporting each class.

4. **Access Control Framework**

   This framework is currently a DIS. This document defines the basic concepts of Access Control and proposes an abstract model for access control, i.e., all actions subject to Access Control must be validated by an Access Enforcement Function (AEF). This function invokes the Access Decision Function which decides if a given action must be carried out or not.

5. **Non-Repudiation Framework**

   This framework describes all aspects of Non-repudiation in Open Systems. This includes the concept of a data recipient being provided with a proof of origin and the concept of a data sender being provided with a proof of delivery.

   It was progressed to CD in December of 1992 (London).

6. **Integrity Framework**

   This framework addresses, mainly, the aspect of data integrity. I.e., ensuring that unauthorized data changes are either not allowed (e.g., Access Control) or detectable (e.g., cryptographic checksums over non-secure media).

   This document was advanced to CD in December 1992 (London).

7. **Confidentiality Framework**

   This framework mirrors the previous one both in scope and status (was advanced to CD in December 1992, London).

   It addresses all aspects of Confidentiality in Open Systems (i.e., mainly how to protect sensitive information cryptographically, by Access Control, or by other means), identifies possible classes of confidentiality mechanisms, defines the services and the abstract data types needed for each Confidentiality mechanism, and addresses the interaction of Confidentiality with other security services and mechanisms.

### WG4: OSI Management

WG4 is currently pursuing work in the following areas:

1. Systems Management Security,
2. Security Audit Trail, and
3. Security for Directories.

In more detail, the standards pursued are:

**OSI Security Management:** currently a Working Draft.

**Security Audit Trail Function:** currently a CD, it is developed as part of OSI Management.

**Security Alarm Reporting Function:** has become an IS.

**Access Control for OSI Applications:** currently a Committee Draft. It addresses Access Control as it pertains to Management.

**Security Audit Trail Framework:** currently a Committee Draft.

**Directory Access Control:** a four part set of Amendments to the OSI Directory standard with all parts presently Proposed Draft Amendments. It should be noted that this Working Group has developed ISO 9594-8 (a joint ISO-CCITT standard), that provides an Authentication framework for the Directory.

### WG6: OSI Session, Presentation and Common Application Services

This working group has already produced an IS addendum *(Association Control Service Element, Authentication)* to ISO 8649. Presently it is working on the *OSI Upper Layer Security Model* (ULSM), a Committee Draft and on a Generic Upper Layer Security Service Element (GULS).

ULSM will specify:

- the security aspects of communication in the Upper Layers of OSI ;
- the support in the Upper Layers of the security services defined in ISO 7498-2 (the OSI Security Architecture) and in the Security Frameworks for Open Systems;
- the positioning of, and relationships among security services and mechanisms in the Upper Layers, according to the guidelines of ISO 7498-2 and ISO/IEC 9545 (ISO 9545 is the *Application Layer Structure* and is also known as CCITT X.207);
- the interactions among the Upper Layers and interactions between the Upper Layers and the Lower Layers, in providing and using security services;
- the requirements for management of security in the Upper Layers, including audit, as guided by the Audit Trail Framework.

An off-shoot of this work is the proposed GULS standard (ISO DIS 11586). GULS, a five part standard, provides security-exchange functions that allow the exchange of security information and security-transformation functions that support the integrity and confidentiality of application data. The latter are supported through ASN.1 extensions.

This work has been the cause of considerable controversy in the past since cryptographic transformations in layer 7 all but replace the functionality of layer 6. But, the appropriate vague statements have been included (i.e., the transformations may be performed in either layer) and the work is now much less controversial. Last minute challenges to this work (including UK proposals for a generic security ESO-OSI abstract interface standard) have not slowed down its progress and it is expected that this work will be incorporated in several applications in the near future.

Most interestingly, the work in GULS is already reflected in the IEEE work on key management. The United States has been interested in having this work accepted as an OSI standard for two reasons:

- Starting with the SDNS series, the United States has always maintained that key management should be done in layer seven, and
- A key management standard in layer seven would greatly simplify the design of the lower layer standards.

A New Work Item (Authentication, Access Control, and key management service elements) has passed ballot in WG6 and the United States intends to port the IEEE work to ISO.

**Other SC21 projects**

WG5 has started a requirements study for Transaction Processing Security and has a new project for FTAM (file transfer) security extensions. Both are new work items that expect to receive support from the security work in WG1 and WG6 and to include GULS or GULS-like features in support of their security needs.

### A.3.5 Security work in SC27

SC27 is the successor of SC20. While, initially, there were doubts about SC27's ability to shake its past, SC27 has initialized promising work and appears to have strong support. SC27 consists of the following three groups:

**WG1** on Generic Security Requirements; its scope covers Security Requirements and Services as well as Guidelines.

**WG2** on Security Mechanisms.

**WG3** on Security Techniques.

WG1's work is of particular importance because the charter of this group includes a key management framework, security information objects, risk analysis, and audit/access control. The key management framework is pursued as a three part standards:

- Overview;
- Key management using symmetric cryptographic techniques;
- Key management using asymmetric techniques.

The last two parts of this standard are developed in WG2 whose program of work includes the following projects :

1. **Modes of Operation for n-Bit Block Cipher Algorithms**, which is a generalization of ISO 8372, *Modes of Operation for 64-Bit Block Cipher Algorithm.*
2. **Entity Authentication Mechanism using an n-bit Secret Key Algorithm**,
3. **Cryptographic Mechanisms for Key Management using Secret Key Techniques**,
4. **Entity Authentication using a Public Key with Two-way and Three-way Handshake**,
5. **Authentication with a Three-way Handshake using Zero-knowledge Techniques**,

6. **Digital Signature Scheme with Message Recovery**,
7. **Hash Functions for Digital Signatures**,
8. **Zero Knowledge Techniques**.

WG3's program of work includes the following:

- Glossary for Computer Security Evaluation Criteria;
- Registry for Functionality Classes;
- Liaison for Common Criteria (see Section 3.1) Editorial Board;
- Evaluation Criteria for Information Technology Security.

### A.3.6 TC68 - Banking and Related Financial Services

TC68 contains two subcommittees whose activities are security relevant:

- **SC2 - Operations and procedures**, and
- **SC6 - Financial Transaction Cards, Related Media and Operations**

Within SC2 the security work is done by WG2, Message Authentication (Security for Wholesale Banking). This Working Group has produced the following ISO standards:

**ISO 8730** *Requirements for Message Authentication*,

**ISO 8731/1** *Approved algorithms for Message Authentication - Part 1 DEA-1 Algorithm.*

**ISO 8731/2** *Approved Algorithms for Message Authentication - Part 2 Message Authentication Algorithm.*

**ISO 8732** *Key Management.*

In addition, it is presently working on the following projects:

1. *Procedures for Message Encipherment - Part 1 General Principles; Part 2 Algorithms.*
2. *Unnumbered Secure Transmission of Personnel Authentication Information and Node Authentication.*
3. *Unnumbered Banking-Key Management - Multiple Centre Environment.*
4. *Data Security Framework for Financial Applications.*

Within SC6, security standards are being developed by WG6, Security in Retail Banking, and WG7, Security Architecture of Banking Systems using the Integrated Circuit Card. WG6 is presently working on the following Standards:

- *Retail Message Authentication*,
- PIN Management and Security; this is a two part standard (*PIN Protection Principles and Techniques* and *Approved Algorithms for PIN Encipherment*).
- *Retail Key Management Standard*.

WG7 is working on a seven part standard on *Financial Transaction Cards*. Its parts are:

- Part 0 - (untitled);
- Part 1 - *Card Life Cycle*, ISO 10202;
- Part 2 - *Transaction Process*;
- Part 3 - *Cryptographic Key Relationships*;
- Part 4 - *Security Application Modules*;
- Part 5 - *Use of Algorithms*; and
- Part 6 - *Cardholder Verification*.

## A.4 CCITT Security Standards

CCITT and ISO have been pursuing several joint security projects. Namely,

- **SG VII Q18 - Message Handling Systems**

  This committee has already produced Recommendations X.400-X.409, *Message Handling Systems*, that also address the security requirements of such systems (most notably X.400, *MHS: System and Service Overview*, and X.402, *MHS: Overall Architecture*, that correspond to ISO 10021-1 and to ISO 10021-2 respectively).

- **SG VII Q19 - Framework for Support of Distributed Applications**

  This group is working jointly with SC21 on the following projects:

  1. *The OSI Security Architecture (ISO 7498-2)*,
  2. *The OSI Security frameworks*,
  3. *The Upper Layer Security model*, and
  4. *A Security model for distributed Applications*.

- **SG VII Q20 - Directory Systems** This group has already finished an authentication framework (for the OSI Directory) and is currently working on access control for the OSI Directory.

- **SG VIII Q28 - Security in Telematic Services**

  This committee is currently preparing a *Security Framework for Telematic Services.*

## A.5 ECMA Security Standards

The European Computer Manufacturers' Association is pursuing the following security related activities:

- **TC29/TGS - Security Aspects of Documents**

  is working on *Security Extensions to ODA* (ODA stands for Office Document Architecture and TC29 works on Document Architecture and Interchange).

- **TC32/TG2 - Distributed Interactive Processing**

  is working on a *Threat/Attack Model for Transaction Processing.*

- **TC32/TG6 - Private Switching Networks**

  is working on integrating cryptography into ISDN.

- **TC32/TG9 - Security in Open Systems**

  TG9 is currently working on an Authentication and Privilege Attribute Security Application and on a protocol to perform Security Association Management, i.e., a key management protocol. It is increasingly apparent that keys are one of the parameters that are necessary for cryptographic services. Thus, key management as a rule involves managing more than just keys.

  It should be noted that this is the ECMA Task Group that works on general aspects of Security and that it has already produced two rather influential documents:

  - ECMA Technical Report 46, *Security in Open Systems - A Security Framework*, and
  - ECMA Standard 138, *Security in Open Systems - Data elements and Service Definitions.*

  This committee includes several experts that are active in ISO and in CCITT. Therefore, both its past and its current work influence the standardization process.

## A.6 IEEE Security Standards

IEEE 802.10 is currently working on a document titled *Standard for Interoperable Local Area Network Security (SILS)* that has eight parts. Part A is the model, part B is Secure Data Exchange (SDE), Part C is Key Management, Part D is Security Management, Part E is SDE of Ethernet, Part F is Sublayer Management, Part G is SDE Security Labels, and Part H is Protocol Information Conformance Statement (PICS) Proforma.

Two things should be noted:

- 802.10 is seeking to expand the ISO 7498-2 Security Architecture so that additional Security Services (i.e., Authentication, Access Control, and Data Integrity) can be provided at layer two.
- X3S3.3, the ANSI committee for lower layers, and SC6 have not supported this project because it not only departs from the familiar ISO Security Architecture but, also, because it incorporates material subject to copyright protection.

In addition, this committee is working on a key management protocol which, while in several ways is a linear descendant of the SDNS work, draws heavily on the work on GULS and on developments within SC27. It is highly probable that this work will form the basis of the ISO key management standard in SC21/WG6.

## A.7 Other Standardization activities

There are several other official, semi-official, and non-official committees and conferences that address security issues. Our purpose is to mention a few of them so as to establish a few players that may affect the standardization process.

One of them is NIST, which has been active in United States Standards activities (often in partnership with DoD agencies) for quite some time. NIST has for several years being hosting the OSE implementors' workshop which includes a special interest group on security. NIST has also being sponsoring workshops on security labels, an area that has direct bearing on both upper and layer protocol standards. As a result of this work, NIST is developing a FIPS on security labels for information transfer [FIP93d].

Another set of organizations are those that were created by the Council of European Community such as ETSI (European Telecommunications Standardization Institute) and EWOS (European Workshop for Open Systems). ETSI and EWOS are producing Functional Standards and have been producing Security profiles for several protocols (such as MHS). Moreover, there are several European collaborative research programs (such as RACE and ESPRIT) that are providing technical support for the European Security Standards.

NATO is sponsoring several standards committees active in security standardization. Thus, a good number of NLSP comments were influenced by NATO concerns.

Finally, interesting and useful work is published as Internet Drafts by the IETF (Internet Engineering Task Force). The most recent draft by John Linn (DEC, member of the common authentication technology wg) is a revival of an older DEC product and defines a Generic Security Service Application Program Interface (GSS-API). Interestingly enough, there have been several recent papers in the UK that advocate this approach as an alternative to the GULS.

IETF has recently formed a new group (IPSEC) tasked with developing an SP3 type protocol in the Internet and a key management protocol.

## A.8 Prospects and Conclusion

This section examines what we can expect in the near and foreseeable future. The short answer is that we shall see a lot of progress, but relatively few Standards.

Indeed, a cursory examination of the standards activities shows that:

1. there is tremendous pressure to develop Security Standards as soon as possible;

2. There are inadequate resources (in part because of the current financial landscape); and

3. even if adequate resources were available, it would take at least three years before a substantial body of work can be completed.

A case that will illustrate the current situation is the need for a key management protocol. The security work in ISO assumes that in most instances cryptographic techniques will be used for security purposes. But, such techniques require shared secrets, such as crypto-keys. Therefore, a key management protocol is a *sine qua non* condition for practically all security protocols, be they upper or lower layer protocols.

If we now look at the work pursued at ISO, we see the following:

- SC27/WG1 is charged with producing a key management framework document. The timetable for this document cannot even be guessed.

- The development of a key management protocol will be done at SC21, presumably, after SC27 produces a mature framework document.

If one now considers any realistic schedule for these events, it appears that there is little or no probability that SC21 will produce stable text for key management within the next two years. This, despite the fact that most of the technical issues for key management that are relevant to this protocol have already been solved elsewhere and there have been proposals (such as the one in the SDNS series) for key management protocols.

Of course, as mentioned earlier, there are strong pressures for developing security standards as early as possible. Already, several standards (Directories, Management, MHS) have invented or are inventing their own security techniques so as to solve urgent problems of their own. At the same time, security protocols such as NLSP and TLSP are inventing ad hoc key management schemes so as to meet their own need for negotiating cryptographic parameters. Other protocols are likely to follow suit unless quick identifiable progress takes place. Therefore, it would appear that those who are interested in Security Standards should follow two, seemingly contradictory policies:

- Bear pressure on the relevant ISO committees; and
- Pursue the development of those Standards as a long term goal. Even if ISO and CCITT decide to act as if business as usual will not do, a good amount of patience will be needed.

Another thing to keep in mind is that however extensive the present program of work seems to be, it will have to be expanded. Experience has shown that the point of greatest vulnerability lies in the areas that are performance bottlenecks. In these areas, such as I/O in Operating Systems which is the door of most penetrations, our need for high performance conflicts with our desire to provide adequately secure mechanisms. The danger of taking potentially disastrous shortcuts is real. Already, there are attempts to add security to the route-construction protocols (which can be seen as layer 3 management protocols). These are attempts to beat back the least sophisticated and the least persistent attacks. No doubt, stronger mechanisms will be needed in the future to protect the protocol that constructs routes and the protocol that forwards data. This is an example of future work that lies just at the periphery of what is presently done. It is quite likely that as we grapple with the security problems that arise in distributed computing, we shall discover the need for additional services and mechanisms and that we will engage in work which today we cannot even imagine.

# Appendix B

# Cryptographic Service Calls

**Shu-jen Chang**

This Appendix describes the cryptographic service calls under development by NIST and proposed to the POSIX.6 Working Group. An overview of this draft API is given in Chapter 5.

The cryptographic service calls are presented in four subsections. Section B.1 describes the databases needed to support the API. Section B.1.1 addresses the database management functions in support of the cryptographic functions. Section B.1.2 presents secret-key cryptographic service calls including message encryption, message integrity, and key management. Similarly, Section B.1.3 addresses the public-key cryptographic service calls including public-key encryption, digital signature, and key management.

In describing the service calls, it should be noted that the specification of the service calls is not tied to any particular programming language. For each service call, the syntax of the call is presented first, followed by its parameter descriptions. Each parameter is listed with its data type and an indication of whether it is an input or output parameter, or both. It is possible for some input parameters to be passed through a trusted path such as a smart card other than from the application programs. For each output parameter, whether it is a single-value parameter or an array of single-value elements, it is assumed that the host application program will allocate the necessary memory storage in advance to receive the output values. The data type of "string" refers to strings of characters or sequences of bytes. Strings are left justified, and padded on the right if necessary. Commands marked with an asterisk are restricted to cryptographic officers (CO).

## B.1 Supporting Cryptographic Databases

Before the cryptographic service calls are presented, it is desirable to address the necessary databases in support of cryptographic functions. To perform cryptographic functions securely, the Cryptographic Module (CM) must exercise proper access control. Users requesting cryptographic services must be authenticated before their secret keys can be retrieved from secure storage. Users' access and usage of data must also be authorized. To support these controls, the following databases must exist.

1. A user authentication database (UDATABASE) must exist. A user's authentication must be verified before making any cryptographic service request. Once verified, a user is considered "logged in" to the CM and a connection is established between the user's host application process and the CM. If multiple users can log in to the CM simultaneously and share its resources, it is the host operating system's and CM's responsibility to maintain the separation of service calls among simultaneous connections. It is therefore assumed that the CM knows the identity of the user executing any CM service call until the user specifically logs out of the CM.

   Each element in the user authentication database should contain at least four fields: user id, user authenticator, user type, and user authorization vector. The user id is simply the user's name. The user authenticator can be a password, a biometrics template, or anything else that can be used to authenticate a user. It may be desirable to control access to the CM through different access privileges. The user having the highest authorization privilege is the cryptographic officer (CO). The field "user type" is used to indicate whether a user is a CO or a regular user. The CO assigns specific cryptographic service calls that a user can access in the user authorization vector. When a CM is used for the first time, a CO should initialize the CM and the UDATABASE would then contain an entry for the CO.

2. For the secret key cryptography, a secure secret key database (SKEYDB) must exist to store the secret keys. Facilities must also be provided to control the lifecycle of a key and ensure that replacement keys are brought into operation securely and old keys are safely destroyed. SKEYDB may contain these fields: user id, key id, key value, key type and/or key attributes, and key counter if applicable. Generally speaking, storage space is more limited in a CM than in a host computer. Therefore, SKEYDB is more likely to reside in the host, even though this is not a requirement. Keys stored outside the CM must be protected by encryption. It is possible and may be desirable to combine UDATABASE and SKEYDB into one single database, which is a design issue to be determined by the implementor.

   For the operation of cryptographic functions, keys must be loaded into the proper registers of the CM before cryptographic functions can take place. These registers are CM-dependent and are not to be confused with the generic secure key database (SKEYDB). The retrieval of keys from SKEYDB and the subsequent loading of the keys into the CM registers is handled by the cryptographic module rather than by the application programs, therefore, no cryptographic service call is defined for key retrieval for the CM. For easy referencing, let the registers of a CM be called CMKEYREG. Depending on the type of CM used, the storage of keys in CMKEYREG may be temporary whereas the storage of keys in SKEYDB is more permanent, that is until a key is specifically removed. Keys can be loaded to or removed from SKEYDB by cryptographic service calls.

3. For the public key cryptography, a separate database (PKEYDB) must exist to store a user's private key and public key certificates. A certificate is associated with a unique identifier (CERTID), which can be a function of the user identification (USERID), the certification authority's identification (CAID), and a certificate serial number (CASERIALNO).

### B.1.1 User Database Management Service Calls

**VERIFYUSER**

| | | |
|---|---|---|
| ( | | |
| UID, | in/out | string |
| LEN, | input | integer |
| UAUTHENT, | input | string |
| RESULT, | output | integer |
| STATUS | output | integer |
| ) | | |

**Parameter Descriptions:**

**UID:** Specifies the address that points to the character string containing the user's identity.

**LEN:** Specifies the length of UAUTHENT in bytes.

**UAUTHENT:** Specifies the address that point to the string of bytes containing the user's authenticator.

**RESULT:** Specifies the address that points to the data storage that will receive the result of the call, which is either 0 or 1.
0 - Pass
1 - Fail

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call verifies the authenticator (UAUTHENT) of length LEN supplied by the UID against the user's authenticator stored in the UDATABASE. A user's identity should be verified before any cryptographic request can be made. The RESULT and STATUS are returned to the host.

***CREATEUSER**

```
(
UID,        in/out  string
UTYPE,      input   character
LEN,        input   integer
UAUTHENT,   in/out  string
STATUS      output  integer
)
```

**Parameter Descriptions:**

**UID:** Specifies the address that points to the character string containing the user's identity.

**UTYPE:** Specifies the user type, for example, "c" for COs, "u" for users.

**LEN:** Specifies the length of UAUTHENT in bytes.

**UAUTHENT:** Specifies the address that points to the string of bytes containing the user's authenticator.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call creates an account for a CO or a user according to the user type indicated (UTYPE). The new account is under the identification of UID. The CO's or the user's authentication information based on UAUTHENT of length LEN is stored in the UDATABASE. It is recommended that SETUSERCOMMAND be called immediately after an account is created. The service call returns the resulting STATUS to the host.

**CHANGEAUTHENT**

| | | |
|---|---|---|
| ( | | |
| OLDLEN, | input | integer |
| OLDAUTHENT, | input | string |
| NEWLEN, | input | integer |
| NEWAUTHENT, | in/out | string |
| STATUS | output | integer |
| ) | | |

**Parameter Descriptions:**

**OLDLEN:** Specifies the length of OLDAUTHENT in bytes.

**OLDAUTHENT:** Specifies the address that points to the string of bytes containing the user's old authenticator.

**NEWLEN:** Specifies the length of NEWAUTHENT in bytes.

**NEWAUTHENT:** Specifies the address that points to the string of bytes containing the user's new authenticator.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
  0 - Service call executed correctly
  $>0$ - Abnormal termination

This service call lets a user change his/her authenticator. If the authenticator (OLDAUTHENT) of length OLDLEN supplied by the user is verified, the user's current authenticator is replaced by NEWAUTHENT of length NEWLEN and the resulting STATUS is returned to the host.

***SETUSERCOMMAND**

(
UID, input string
AV, input string
STATUS output integer
)

| | | |
|---|---|---|
| UID, | input | string |
| AV, | input | string |
| STATUS | output | integer |

**Parameter Descriptions:**

**UID:** Specifies the address that points to the character string containing the user's identity.

**AV:** Specifies the address that points to the string of bytes containing the authorization vector. An authorization vector defines the service calls that a user can access. Each bit within the byte in the authorization vector corresponds to a service call. A one in the bit enables the corresponding service call whereas a zero disables it. For example, the correspondence between the service calls and their bit positions for the first byte of AV looks as follows:

Bit 0 - VERIFYUSER
Bit 1 - CREATEUSER
Bit 2 - CHANGEAUTHENT
Bit 3 - SETUSERCOMMAND
Bit 4 - SHOWUSERCOMMAND
Bit 5 - DELETEUSER
Bit 6 - LOGOUT
Bit 7 - ENCIPHER

It is assumed that a list of the service calls is available to the CO.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

0 - Service call executed correctly
$>0$ - Abnormal termination

This service call lets the CO set specific service calls that a user (UID) can access. The authorization vector (AV) for user UID is stored in the UDATABASE, and the resulting STATUS is returned to the host.

**SHOWUSERCOMMAND**

(

| | | |
|---|---|---|
| UID, | input | string |
| AVLEN, | input | integer |
| AV, | output | string |
| STATUS | output | integer |

)

**Parameter Descriptions:**

**UID:** Specifies the address that points to the character string containing the user's identity if the service call is executed by a CO; null otherwise.

**AVLEN:** Specifies the total number of cryptographic service calls defined. Since each service call is represented by one bit in AV as described in SETUSERCOMMAND, this parameter indicates how many bits of AV to read which are meaningful.

**AV:** Specifies the address that points to the string of bytes containing the authorization vector associated with the user. "One" bits indicate enabled service calls whereas "zero" bits indicate disabled service calls.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

0 - Service call executed correctly

$>0$ - Abnormal termination

This service call uses AVLEN to determine how many bits of the authorization vector (AV) of UID is to be read, and returns the AV and resulting STATUS to the host.

***DELETEUSER**

(
UID, input string
STATUS output integer
)

**Parameter Descriptions:**

**UID:** Specifies the address that points to the character string containing the name of the user whose record is to be removed from UDATABASE.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

0 - Service call executed correctly
$>0$ - Abnormal termination

This service call allows a CO to remove user UID's entry from the UDATABASE. Every field in the database pertaining to the user is deleted and the storage is freed up. It should be noted that DELETEKEY may need to be called before DELETEUSER so that the user's keys are removed from SKEYDB before the user's account is closed. The resulting STATUS is returned to the host.

**LOGOUT**

(
STATUS output integer
)

**Parameter Descriptions:**

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

0 - Service call executed correctly
$>0$ - Abnormal termination

This service call allows the user currently logged on to the CM to log out of the CM and returns the STATUS to the host.

### B.1.2 Secret Key Cryptography Service Calls

**Encryption and Data Integrity Service Calls**

**ENCIPHER**

(

| | | |
|---|---|---|
| ALGID, | input | integer |
| MODE, | input | integer |
| PLEN, | input | integer |
| PT, | input | string |
| KEYID, | input | string |
| IV, | in/out | string |
| NBITFB, | input | integer |
| CHAIN, | input | integer |
| CLEN, | output | integer |
| CT, | output | string |
| STATUS | output | integer |

)

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for enciphering.
- 1 - DES
- >1 - Reserved for later use

**MODE:** Specifies the mode of the enciphering operation.
- 1 - Electronic Code Book
- 2 - Cipher Block Chaining
- 3 - Cipher Feedback
- 4 - Output Feedback

**PLEN:** Specifies the length of the plaintext data in bytes.

**PT:** Specifies the address that points to the string of bytes containing the plaintext data.

**KEYID:** Specifies the address that points to the character string containing the name of the encrypting key.

**IV:** Specifies the address that points to the string of bytes containing the 8-byte initialization vector. Used in modes 2, 3, or 4. Null otherwise.

**NBITFB:** An integer between 1 and 64 indicating the number of bits of feedback to use in Cipher Feedback or Output Feedback mode. 0 in other cases.

**CHAIN:** Specifies if chaining of consecutive encryption is desired. If chaining is desired, intermediate data values should be preserved across calls. This is useful for encrypting large files.

- 0 - PT is the only block to be encrypted, i.e., it is the first and the last block.
- 1 - First block, but not the last.
- 2 - Middle blocks, i.e., not first, not last.
- 3 - Last block.

**CLEN:** Specifies the length of the ciphertext in bytes.

**CT:** Specifies the address that points to the string of bytes containing the ciphertext. Since CT is likely to contain nonprintable characters, it is necessary to use other routines to convert the string of packed bytes into a string of ASCII hexadecimal characters when printing out the content of CT.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

- 0 - Service call executed correctly
- 1 - CT string size overflow
- >1 - Other abnormal termination

This service call enciphers plaintext data (PT) of length PLEN in the specified algorithm (ALGID) and MODE using KEYID as the encryption key. For modes 2, 3, and 4, an initialization vector may be specified in the IV parameter. For Cipher Feedback and Output Feedback Modes, NBITFB specifies the number of bits of feedback to use. The ciphertext (CT), the length of the ciphertext (CLEN), and the STATUS are returned to the host. Depending on the mode of operation, some padding may be added to the input plaintext data for a 64-bit block cipher, hence the length of the ciphertext (CLEN) may be greater than the length of the plaintext (PLEN). If STATUS indicates a condition of string size overflow of the ciphertext (CT), the output parameter CLEN should indicate the length of the ciphertext and the host should increase the memory storage allocated for CT accordingly. When encrypting a large file, there may not be enough memory to hold the entire file, in this case, a means for chaining consecutive requests for multiple blocks is provided by the CHAIN parameter. Depending on the value of this parameter, the CM would know when and when not to preserve intermediate values. If chaining is desired, the CM should preserve intermediate values. The distinction between the first block (CHAIN set to 1) and the intermediate blocks (CHAIN set to 2) can provide helpful information for the CM to implement the service call efficiently, since the first block usually requires initial setup which may not be needed for intermediate blocks.

**DECIPHER**

```
(
ALGID,   input   integer
MODE,    input   integer
CLEN,    input   integer
CT,      input   string
KEYID,   input   string
IV,      input   string
NBITFB,  input   integer
CHAIN,   input   integer
PLEN,    output  integer
PT,      output  string
STATUS   output  integer
)
```

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for deciphering.
- 1 - DES
- >1 - Reserved for later use

**MODE:** Specifies the mode of the deciphering operation.
- 1 - Electronic Code Book
- 2 - Cipher Block Chaining
- 3 - Cipher Feedback
- 4 - Output Feedback

**CLEN:** Specifies the length of the ciphertext in bytes.

**CT:** Specifies the address that points to the string of bytes containing the ciphertext. CT may contain nonprintable characters.

**KEYID:** Specifies the address that points to the character string containing the name of the decrypting key.

**IV:** Specifies the address that points to the string of bytes containing the 8-byte initialization vector for modes 2, 3, or 4. Null otherwise.

**NBITFB:** An integer between 1 and 64 indicating the number of bits of feedback to use for Cipher Feedback Mode or Output Feedback Mode. 0 for other cases.

**CHAIN:** Specifies if chaining of consecutive decryptions is desired. If chaining is desired, intermediate data values should be preserved across calls. This is useful for decrypting large files.

0 - CT is the only block to be decrypted, i.e., it is the first and the last block.
1 - First block, but not the last.
2 - Middle blocks, i.e., not first, not last.
3 - Last block.

**PLEN:** Specifies the length of the plaintext in bytes.

**PT:** Specifies the address that points to the string of bytes containing the plaintext data.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.

0 - Service call executed correctly
1 - PT string size overflow
>1 - Abnormal termination

This service call decrypts the ciphertext (CT) of length CLEN in the specified algorithm (ALGID) and MODE using KEYID as the decrypting key. The input parameter IV specifies the initialization vector for modes 2, 3, and 4. For Cipher Feedback and Output Feedback modes, NBITFB specifies the number of bits of feedback to use. The decrypted plaintext (PT), the length of the plaintext (PLEN), and the resulting STATUS are returned to the host. The chaining parameter (CHAIN) chains consecutive decryption requests for multiple blocks. Depending on the value of the parameter, the CM would know when and when not to preserve intermediate values across calls.

**COMPUTEDAC**

```
(
ALGID,  input   integer
LEN,    input   integer
DATA,   input   string
KEYID,  input   string
CHAIN,  input   integer
DAC,    output  string
STATUS  output  integer
)
```

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for COMPUTEDAC.

1 - DES

>1 - Reserved for later use

**LEN:** Specifies the length of the data in bytes.

**DATA:** Specifies the address that points to the string of bytes containing the data whose Data Authentication Code (DAC) is to be computed.

**KEYID:** Specifies the address that points to the character string containing the name of the key used for DACing.

**CHAIN:** Specifies if chaining of consecutive DAC operations is desired. If chaining is desired, intermediate data values should be preserved across calls.

0 - DATA points to the only block whose DAC is to be computed.

1 - DATA points to the First block, but not the last block.

2 - DATA points to a middle block.

3 - DATA points to the last block.

**DAC:** Specifies the address that points to the string of packed bytes that will receive the computed DAC. Since DAC is likely to contain nonprintable characters, it is necessary to use another routine to convert the string of packed bytes into a string of ASCII hexadecimal characters before the content of DAC can be printed.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.

0 - Service call executed correctly

>0 - Abnormal termination

This service call computes a Data Authentication Code (DAC) on the DATA of indicated LEN using KEYID as the encrypting key. The computed DAC and resulting STATUS are returned to the host. Chaining of consecutive DAC requests is provided by the CHAIN parameter. If chaining is desired, the CM should preserve intermediate data values across consecutive calls.

**VERIFYDAC**

```
(
ALGID,   input   integer
LEN,     input   integer
DATA,    input   string
KEYID,   input   string
DAC,     input   string
CHAIN,   input   integer
RESULT,  output  integer
STATUS   output  integer
)
```

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for VERIFYDAC.

1 - DES
>1 - Reserved for later use

**LEN:** Specifies the length of the data in bytes.

**DATA:** Specifies the address that points to the string of bytes containing the data whose DAC is to be verified.

**KEYID:** Specifies the address that points to the character string containing the name of the key used for DACing.

**DAC:** Specifies the address that points to the string of bytes containing the input Data Authentication Code. If the user-entered Data Authentication Code is a string of ASCII hexidecimal characters with a blank space separating the left half and the right half of the code, it should be converted to a string of packed bytes first before calling VERIFYDAC.

**CHAIN:** Specifies if chaining of consecutive calls is desired. If chaining is desired, intermediate data values should be preserved across calls.

0 - DATA points to the only block whose DAC is to be verified.
1 - DATA points to the first block, but not the last block.
2 - DATA points to a middle block.
3 - DATA points to the last block.

**RESULT:** Specifies the address that points to the data storage that will receive the result of DAC verification.

0 - DAC is verified
1 - DAC is not verified

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.

0 - Service call executed correctly

$>0$ - Abnormal termination

This service call computes a Test Data Authentication Code (TDAC) on the DATA of indicated LEN using KEYID as the encrypting key, and checks if TDAC matches the input DAC. The RESULT and STATUS are returned to the host. Chaining of consecutive VERIFYDAC requests is provided by the chaining parameter (CHAIN). If chaining is used, the CM should preserve intermediate data values across calls.

**Key Management Service Calls**

**GENKEY**

```
(
KEYID,          input    string
LEN,            input    integer
OUTPUTCLEAR,    input    integer
ATTRIB,         input    unsigned character
PTKEY,          output   string
STATUS          output   integer
)
```

**Parameter Descriptions:**

**KEYID:** Specifies the address that points to the character string containing the name of the key to be generated.

**LEN:** Specifies the length of key in bits. DES keys are 64 bits for a single key, 128 bits for a key pair.

**OUTPUTCLEAR:** Specifies if the generated plaintext key should be returned to the host.
0 - Do not return the plaintext key to the host.
1 - Return the plaintext key to the host.

**ATTRIB:** Represented in a byte, this parameter specifies the operations that the generated key can be used for (See SETATTRIB service call for reference). Each bit within the byte corresponds to an operation, a one in the bit enables the operation whereas a zero disables the operation. The assigned bit positions for the basic key operations are as follows:

Bit 0 - Encryption
Bit 1 - Decryption
Bit 2 - DAC Generation
Bit 3 - DAC Verification
Bit 4 - Key exportable outside CM
Bit 5 - Attribute Lock Bit
Bit 6 - Not used
Bit 7 - Not used

**PTKEY:** Specifies the address that points to the string of bytes containing the generated plaintext key if OUTPUTCLEAR is set; null otherwise.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.
0 - Service call executed correctly
$>0$ - Abnormal termination

This service call generates a secret key of odd parity of length LEN and stores it under the name of KEYID in SKEYDB. If OUTPUTCLEAR is set, the plaintext key (PTKEY) and STATUS are returned to the host; otherwise, PTKEY contains a null pointer and only STATUS is returned. The key may be initialized with a key counter using the SETCOUNT service call. Allowable operations for the key are set according to what is specified in ATTRIB, the key attributes.

**DELETEKEY**

| | | |
|---|---|---|
| ( | | |
| UID, | input | string |
| KEYID, | input | string |
| STATUS | output | integer |
| ) | | |

**Parameter Descriptions:**

**UID:** Specifies the address that points to the character string containing the user's identity whose key is to be deleted from SKEYDB.

**KEYID:** Specifies the address that points to the character string containing the name of the key to be deleted.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.
0 - Service call executed correctly
$>0$ - Abnormal termination

This service call deletes the key identified by KEYID of user UID from SKEYDB, and returns the resulting STATUS to the host. A user can only delete his/her own keys, however, a CO may delete a user's keys before closing a user's account.

**LOADKEY**

```
(
KEYID,   input   string
LEN,     input   integer
KEY,     input   string
PARITY,  input   integer
STATUS   output  integer
)
```

**Parameter Descriptions:**

**KEYID:** Specifies the address that points to the character string containing the name of the key to be loaded.

**LEN:** Specifies the length of key in bits. DES keys are 64 bits for a single key, 128 bits for a key pair.

**KEY:** Specifies the address that points to the string of bytes containing the clear key value to be loaded.

**PARITY:** Specifies if odd parity is to be used for the key to be loaded.
- 0 - Ignore parity checks
- 1 - Set odd parity

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.
- 0 - Service call executed correctly
- $>0$ - Abnormal termination

This service call loads a clear KEY of length LEN to SKEYDB under the identity of KEYID. If PARITY is set, the key is set to odd parity; otherwise, the parity of the key is not checked. The resulting STATUS is returned to the host.

**EXPORTKEY**

```
(
KEYID,          input   string
KKID,           input   string
NOS,            input   integer
KOFFSET,        input   integer
ORI,            input   string
RCV,            input   string
LEN,            output  integer
ENCRYPTEDKEY,   output  string
CTT,            output  string
STATUS          output  integer
)
```

**Parameter Descriptions:**

**KEYID:** Specifies the address that points to the character string containing the name of the key to be exported.

**KKID:** Specifies the address that points to the character string containing the name of the key encrypting key.

**NOS:** Specifies if notarization is desired or not.
0 - Notarization not desired
1 - Notarization desired

**KOFFSET:** Specifies if key offset is to be used.
0 - Key offset not desired
1 - Key offset desired

**ORI:** Specifies the address that points to the character string containing the identity of the message originator.

**RCV:** Specifies the address that points to the character string containing the identity of the message recipient.

**LEN:** Specifies the address that points to the data storage that will receive the length of the encrypted key (ENCRYPTEDKEY) in bits. DES keys are 64 bits for a single- length key, 128 bits for a key pair.

**ENCRYPTEDKEY:** Specifies the address that points to the string of bytes containing the encrypted key value of KEYID.

**CTT:** Specifies the address that points to a string of 7 bytes containing the transmit count if NOS or KOFFSET is used; null otherwise.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.

- 0 - Service call executed correctly
- $>0$ - Abnormal termination

This service call encrypts a plaintext key before exporting the key outside the CM. If notarization (NOS) is desired, a notarizing key is formed inside the CM before it is used to encrypt the key value of KEYID. A notarizing key is formed by XORing the plaintext key value of KKID with a notary seal formed from the transmit count (CTT) of KKID and the identities of the message originator (ORI) and the intended recipient (RCV). (See ANSI X9.17 Standard for reference.) If key offset is desired, the plaintext value of KKID is XORed with the transmit count (CTT) of KKID before the result is used to encrypt the key value of KEYID (See ANSI X9.17 Standard for reference). The notarization flag (NOS) and the key offset flag (KOFFSET) are mutually exclusive, i.e., both flags can not be set to 1 in the same call. If neither key offset nor notarization is desired, the key value of KEYID is simply encrypted by the key value of KKID, and the ORI and RCV fields are ignored. The length of the encrypted key (LEN), the encrypted key (ENCRYPTEDKEY) itself, and the resulting STATUS are returned to the host. If notarization or key offset is used, CTT is also returned to the host; otherwise, a null pointer is returned.

**IMPORTKEY**

```
(
KEYID,          input   string
KKID,           input   string
LEN,            input   integer
ENCRYPTEDKEY,   input   string
NOS,            input   integer
KOFFSET,        input   integer
ORI,            input   string
RCV,            input   string
CTR,            input   string
PARITY,         output  integer
STATUS          output  integer
)
```

**Parameter Descriptions:**

**KEYID:** Specifies the address that points to the character string containing the name of the key to be imported.

**KKID:** Specifies the address that points to the character string containing the name of the key used to encrypt KEYID.

**LEN:** Specifies the length of the key to be imported in bits. DES keys are 64 bits for a single key, 128 bits for a key pair.

**ENCRYPTEDKEY:** Specifies the address that points to the string of bytes containing the encrypted key value of KEYID.

**NOS:** Specifies if notarization is to be used.
0 - No notarization used
1 - Notarization used

**KOFFSET:** Specifies if key offset is to be used.
0 - Key offset not desired
1 - Key offset desired

**ORI:** Specifies the address that points to the character string containing the identity of the message originator.

**RCV:** Specifies the address that points to the character string containing the identity of the message recipient.

**CTR:** Specifies the address that points to a string of 7 bytes containing the receive count used in key notarization.

**PARITY:** Specifies if the imported key conformed to odd parity.

- 0 - Imported key not conformed to odd parity
- 1 - Imported key conformed to odd parity

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.

- 0 - Service call executed correctly
- $>0$ - Abnormal termination

This service call decrypts an imported key and stores it in SKEYDB. If notarization (NOS) was used, a notarizing key was formed by XORing the key value of the key encrypting key (KKID) with a notary seal formed from the receive count (CTR) of KKID and the identities of the message originator (ORI) and the recipient (RCV). The notarizing key is then used to decrypt the encrypted key (ENCRYPTEDKEY). (For the processing of key counters and the notarization procedure, see the ANSI X9.17 Standard for reference.) If key offset was used, the key value of KKID is XORed with the receive count (CTR) of KKID before the result is used to decrypt the ENCRYPTEDKEY. The CTR is always compared with the stored receive count and processed according to the ANSI X9.17 standard. The notarization flag (NOS) and the key offset flag (KOFFSET) are mutually exclusive, i.e., both can not be set to 1 in the same call. If neither notarization nor key offset was used, the ORI, RCV, and COUNT fields are ignored and the ENCRYPTEDKEY of length LEN is simply decrypted by KKID. The deciphered key is checked for odd parity and stored in SKEYDB under the identity of KEYID. The result of the parity check and the STATUS of processing the call are returned to the host.

**SETATTRIB**

| | | |
|---|---|---|
| ( | | |
| KEYID, | input | string |
| ENCRYPTEDATTRIB, | input | string |
| STATUS | output | integer |
| ) | | |

**Parameter Descriptions:**

**KEYID:** Specifies the address that points to the character string containing the name of the key whose attributes are to be set.

**ENCRYPTEDATTRIB:** Specifies the address that points to a string of 8 bytes containing the encrypted attributes of KEYID. ENCRYPTEDATTRIB is encrypted under the key value of KEYID.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.
0 - Service call executed correctly
$>0$ - Abnormal termination

This service call sets the operations that a key is allowed to perform, referred to as a key's attributes, according to what is specified in ENCRYPTEDATTRIB. Currently six operations are defined. The key attributes are represented in a byte with the assigned bit positions:

Bit 0 - Encryption
Bit 1 - Decryption
Bit 2 - DAC Generation
Bit 3 - DAC Verification
Bit 4 - Key exportable out of the CM
Bit 5 - Attribute Lock Bit
Bit 6 - Currently not used
Bit 7 - Currently not used

A set bit (i.e., bit set to 1) within the byte enables the specific operation for the key. For example, if Bit 4 is set, KEYID can be exported outside the CM using the EXPORTKEY service call. If enabled, the lock bit (Bit 5) locks the key's attributes and makes it impossible to change the attributes afterwards. Before calling SETATTRIB, the plaintext key attribute should have been padded on the right with seven zero bytes and DES-encrypted under the key value of KEYID. Upon receiving the SETATTRIB call, the CM decrypts ENCRYPTEDATTRIB (with the key value of KEYID) and sets the attributes for KEYID. It is the responsibility of the CM to check the key attributes before a key is used for any operation. The service call returns the STATUS of the call to the host.

**READATTRIB**

```
(
KEYID,    input   string
ATTRIB,   output  unsigned character
STATUS    output  integer
)
```

**Parameter Descriptions:**

**KEYID:** Specifies the address that points to the character string containing the name of the key whose attributes are to be retrieved.

**ATTRIB:** Specifies the address that points to the one-byte data storage that will receive the attributes of KEYID.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.
- 0 - Service call executed correctly
- $>0$ - Abnormal termination

This service call reads the attributes associated with KEYID, stores the retrieved attributes in ATTRIB and returns the STATUS to the host. Refer to the SETATTRIB service call for the format of key attributes.

**XORKEYS**

```
(
NEWKEYID,  input   string
KEYID1,    input   string
KEYID2,    input   string
STATUS     output  integer
)
```

**Parameter Descriptions:**

**NEWKEYID:** Specifies the address that points to the character string containing the name of the new key formed by XORing the keys of KEYID1 and KEYID2.

**KEYID1:** Specifies the address of the character string containing the name of the key to be XORed with the key of KEYID2.

**KEYID2:** Specifies the address of the character string containing the name of the key to be XORed with the key of KEYID1.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.
- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call exclusive-ors two keys in the SKEYDB identified by KEYID1 and KEYID2 to form a new key identified by NEWKEYID. The service call facilitates the use of dual control in forming a working key. The attributes for the new key are set equal to the intersection of the attributes possessed by KEYID1 and KEYID2.

**SETCOUNT**

```
(
KEKID,  input   string
CTT,    input   string
CTR,    input   string
STATUS  output  integer
)
```

**Parameter Descriptions:**

**KEKID:** Specifies the address that points to the character string containing the name of the key encrypting key whose counters are to be reset.

**CTT:** Specifies the address that points to a string of 7 bytes containing the transmit count to be set for KEKID.

**CTR:** Specifies the address that points to a string of 7 bytes containing the receive count to be set for KEKID.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.
- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call lets the host set the transmit count (CTT) and the receive count (CTR) associated with KEKID in the SKEYDB. The resulting STATUS is returned to the host. The SETCOUNT and READCOUNT service calls are added for compatibility with the ANSI X9.17 Key Management (Wholesale) Standard. The initial values of key counters should be selected according to specifications in the X9.17 Standard.

**READADCOUNT**

```
(
KEKID,  input   string
CTT,    output  string
CTR,    output  string
STATUS  output  integer
)
```

**Parameter Descriptions:**

**KEKID:** Specifies the address that points to the character string containing the name of the key encrypting key whose counters are to be retrieved.

**CTT:** Specifies the address that points to a string of 7 bytes containing the transmit count of KEKID.

**CTR:** Specifies the address that points to a string of 7 bytes containing the receive count of KEKID.

**STATUS:** Specifies the address that points to the data storage that will receive the status of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call lets the host read the transmit count (CTT) and the receive count (CTR) associated with KEKID in the SKEYDB. The counters and the resulting STATUS are returned to the host. SETCOUNT and READCOUNT service calls are added for compatibility with the ANSI X9.17 Key Management (Wholesale) Standard.

### B.1.3 Public Key Cryptography Service Calls

**Encryption and Digital Signature Service Calls**

**PUBENCIPHER**

```
(
ALGID,          input   integer
MODULUS_SIZE,   input   integer
RCVR_PUBKEY,    input   string
PLEN,           input   integer
PT,             input   string
CLEN,           output  integer
CT,             output  string
STATUS          output  integer
)
```

**Parameter Descriptions:**

**ALGID:** Specifies the encryption algorithm to be used:
- 1 - RSA
- >1 - Reserved for future use

**MODULUS_SIZE:** Specifies the size of the key modulus in bytes.

**RCVR_PUBKEY:** Specifies the address that points to the string of bytes containing the public key of the intended recipient of the enciphered message.

**PLEN:** Specifies the length of the plaintext data in bytes. For the RSA encryption algorithm, PLEN should be no greater than MODULUS_SIZE.

**PT:** Specifies the address that points to the string of bytes containing the plaintext data. For the RSA encryption algorithm, the binary value of PT must be less than the binary value of the key modulus.

**CLEN:** Specifies the length of the ciphertext in bytes. For the RSA encryption algorithm, CLEN would be no greater than MODULUS_SIZE.

**CT:** Specifies the address that points to the string of bytes containing the ciphertext. For the RSA encryption algorithm, the binary value of CT would be less than the binary value of the key modulus.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call uses the encryption algorithm specified in ALGID to encipher a message. Currently only the RSA algorithm supports public-key encryption. The size of the key modulus is specified in MODULUS_SIZE. The intended recipient's public key (RCVR_PUBKEY) is used to encipher the plaintext message (PT) of length PLEN. The resulting ciphertext (CT), its length (CLEN), and the STATUS of the call are returned to the host.

**PUBDECIPHER**

```
(
ALGID,            input    integer
MODULUS_SIZE,     input    integer
RCVR_PRIKEYID,    input    string
CLEN,             input    integer
CT,               input    string
PLEN,             output   integer
PT,               output   string
STATUS            output   integer
)
```

**Parameter Descriptions:**

**ALGID:** Specifies the decryption algorithm to be used:

- 1 - RSA
- >1 - Reserved for future use

**MODULUS_SIZE:** Specifies the size of the key modulus in bytes.

**RCVR_PRIKEYID:** Specifies the address that points to the character string containing the identity of the message recipient's private key.

**CLEN:** Specifies the length of the ciphertext in bytes. For the RSA algorithm, CLEN should be no greater than MODULUS_SIZE.

**CT:** Specifies the address that points to the string of bytes containing the ciphertext. For the RSA algorithm, the binary value of CT should be less than the binary value of the key modulus.

**PLEN:** Specifies the length of the plaintext data in bytes. For the RSA algorithm, PLEN would be no greater than MODULUS_SIZE.

**PT:** Specifies the address that points to the string of bytes containing the plaintext data. For the RSA algorithm, the binary value of PT would be less than the binary value of the key modulus.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call uses the algorithm specified in ALGID to decipher a message. Currently only the RSA algorithm supports public-key decryption. The size of the key modulus is specified in MODULUS_SIZE. The message recipient's private key identified by RCVR_PRIKEYID is used to decipher the ciphertext (CT) of length CLEN. The resulting plaintext (PT), its length (PLEN), and the STATUS of the call are returned to the host.

**SIGN**

```
(
ALGID,       input   integer
LEN,         input   integer
DATA,        input   string
MDID,        input   string
PRIKEYID,    input   string
SIGNLEN,     output  integer
SIGNATURE,   output  string
STATUS       output  integer
)
```

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for enciphering:
- 1 - RSA
- 2 - DSA
- 3 - El Gamal Signature Scheme
- >3 - Reserved for future use

**LEN:** Specifies the length of the data in bytes

**DATA:** Specifies the address that points to the string of bytes containing the data to be processed

**MDID:** Specifies the message digest algorithm used for producing the message digest:
- 1 - MD2
- 2 - MD4
- 3 - SHA
- >3 - Reserved for future use

**PRIKEYID:** Specifies the address that points to the character string containing the identity of the private key associated with the signer.

**SIGNLEN:** Specifies the length of the signature in bytes

**SIGNATURE:** Specifies the address that points to the string of bytes containing the result of applying the private key to the data.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call applies a message digest algorithm, specified by MDID, to the DATA of indicated LEN resulting in a message digest. Note that the DATA must be left justified (from the least significant byte to the most significant byte) and right padded with zeroes. The resulting message digest is used to compute a SIGNATURE, based on the ALGID specified, by applying the private key associated with PRIKEYID to the message digest.Note when a modulus is used, the message digest must be less than or equal to the modulus associated with the signature algorithm specified by ALGID. The service call returns the resulting SIGNATURE, SIGNLEN, and STATUS to the host.

NOTE1: Information such as USERID, CAID, and CASERIALNO could be in the data or sent separately in order to indicate the correct public key, PUBKEY, used to verify the signature.

NOTE2: When it is desired to compute the signature on the DATA of indicated LEN without applying a hashing function, use PUBDECIPHER.

**VERIFYSIGNATURE**

(

| | | |
|---|---|---|
| ALGID, | input | integer |
| MDID, | input | integer |
| SIGNLEN, | input | integer |
| SIGNATURE, | input | string |
| CERTID, | input | string |
| LEN, | input | integer |
| DATA, | input | string |
| RESULT, | output | integer |
| STATUS | output | integer |

)

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for enciphering:

1 - RSA
2 - DSA
3 - El Gamal Signature Scheme
$>3$ - Reserved for future use

**MDID:** Specifies the message digest algorithm used for producing the message digest:

1 - MD2
2 - MD4
3 - SHA
$>3$ - Reserved for future use

**SIGNLEN:** Specifies the length of the signature in bytes

**SIGNATURE:** Specifies the address that points to the string of bytes containing the result of applying the private key to the data.

**CERTID:** Specifies the address that points to the character string containing the identity of the certificate

**LEN:** Specifies the length of the data in bytes

**DATA:** Specifies the address that points to the string of bytes containing the data to be processed

**RESULT:** Specifies the address that points to the data storage that will receive the result of the call, which is either 0 or 1

0 - Pass
1 - Fail

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

- 0 - Service call executed correctly
- $>0$ - Abnormal termination

Based on the ALGID specified, this service call verifies the SIGNATURE of indicated SIGNLEN by applying the public key obtained from the certificate associated with CERTID, to the SIGNATURE to reveal a Test Message Digest (TMD). It sets the correct RESULT to indicate if TMD is identical with the Message Digest (MD) computed by applying a message digest algorithm, specified by MDID, to the DATA of indicated LEN. Note when a modulus is used, the message digest must be less than or equal to the modulus of the signature algorithm, ALGID. The RESULT and STATUS are returned to the host.

NOTE: When it is desired to verify the signature on the DATA of indicated LEN without applying a hashing function, use PUBENCIPHER.

**Key Management Service Calls**

**GENPUBKEYPAIR**

(

| | | |
|---|---|---|
| ALGID, | input | integer |
| ENCRYPTEXP, | input | integer |
| LEN, | input | integer |
| PRIKEYID, | input | string |
| PUBKEY, | output | string |
| STATUS | output | integer |

)

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for enciphering:

1 - RSA
2 - El Gamal Signature Scheme
3 - Diffie/Hellman
>3 - Reserved for future use

**ENCRYPTEXP:** Specifies the encryption exponent used:

1 - Provided by the system
>0 - Provided by the user

**LEN:** Specifies the length of the keys in bits

**PRIKEYID:** Specifies the address that points to the character string containing the identity of the private key.

**PUBKEY:** Specifies the address that points to the string of bytes containing the public key data.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

0 - Service call executed correctly
>0 - Abnormal termination

Based on the ALGID and ENCRYPTEXP specified, this service call generates a public/private key pair of length LEN indexed by the user identification known by the host. The private key is stored in secure memory as PRIKEYID. The service call returns the PUBKEY and the resulting STATUS to the host.

**STORECERTIFICATE**

```
(
CERTLEN,        input   integer
CERTIFICATE,    input   string
CERTID,         output  string
STATUS          output  integer
)
```

**Parameter Descriptions:**

**CERTLEN:** Specifies the length of the certificate in bytes

**CERTIFICATE:** Specifies the address that points to the string of bytes containing the signed data item produced when a Certification Authority representing an organization applies a digital signature to a collection of data consisting of, at minimum, the following information: USERID, CAID, CASERIALNO, PUBKEY, EXPDATE, ALGID.

**CERTID:** Specifies the address that points to the character string containing the identity of the certificate

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call stores the contents of CERTIFICATE of length CERTLEN in the Cryptographic Module (CM) under the identity of CERTID and returns the resulting STATUS to the host.

**RETRIEVECERTIFICATE**

```
(
CERTID,        input   string
CERTLEN,       output  integer
CERTIFICATE,   output  string
STATUS         output  integer
)
```

**Parameter Descriptions:**

**CERTID:** Specifies the address that points to the character string containing the identity of the certificate

**CERTLEN:** Specifies the length of the certificate in bytes

**CERTIFICATE:** Specifies the address that points to the string of bytes containing the signed data item produced when a Certification Authority representing an organization applies a digital signature to a collection of data consisting of, at minimum, the following information: USERID, CAID, CASERIALNO, PUBKEY, EXPDATE, ALGID.

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call retrieves the CERTIFICATE identified by CERTID from the Cryptographic Module (CM). It returns the CERTIFICATE, the length of the certificate CERTLEN, and the resulting STATUS to the host.

**DELETEPRIKEY**

(
PRIKEYID, input string
STATUS output integer
)

**Parameter Descriptions:**

**PRIKEYID:** Specifies the address that points to the character string containing the identity of the private key

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call allows the private key associated with PRIKEYID to be deleted by the owner of that key. The service call returns the resulting STATUS to the host.

***DELETECERTIFICATE**

(
CERTID, input string
STATUS output integer
)

**Parameter Descriptions:**

**CERTID:** Specifies the address that points to the character string containing the identity of the certificate

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
0 - Service call executed correctly
>0 - Abnormal termination

This service call deletes the certificate associated with CERTID. The service call returns the resulting STATUS to the host.

**PUBEXPORTKEY**

```
(
ALGID,          input   integer
KEYID,          input   string
CERTID,         input   string
ENCRYPTEDKEY,   output  string
STATUS          output  integer
)
```

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for enciphering:

- 1 - RSA
- 2 - El Gamal Signature Scheme
- 3 - Diffie/Hellman
- >3 - Reserved for future use

**KEYID:** Specifies the address that points to the character string containing the name of the key to be exported

**CERTID:** Specifies the address that points to the character string containing the identity of the certificate

**ENCRYPTEDKEY:** Specifies the address that points to the string of bytes containing the encrypted key value of KEYID

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.

- 0 - Service call executed correctly
- >0 - Abnormal termination

This service call uses the ALGID specified along with the public key obtained from the certificate associated with CERTID from the Cryptomodule's Database and uses this key to RSA encrypt the key associated with KEYID.It returns the ENCRYPTEDKEY and the resulting STATUS to the host.

**PUBIMPORTKEY**

```
(
ALGID,          input   integer
KEYID,          input   string
PRIKEYID,       input   string
ENCRYPTEDKEY,   input   string
STATUS          output  integer
)
```

**Parameter Descriptions:**

**ALGID:** Specifies the algorithm used for enciphering:
- 1 - RSA
- 2 - El Gamal Signature Scheme
- 3 - Diffie/Hellman
- $>3$ - Reserved for future use

**KEYID:** Specifies the address that points to the character string containing the name of the key to be imported

**PRIKEYID:** Specifies the address that points to the character string containing the identity of the private key used to decipher KEYID

**ENCRYPTEDKEY:** Specifies the address that points to the string of bytes containing the encrypted key value of KEYID

**STATUS:** Specifies the address that points to the data storage that will receive the result of processing the service call.
- 0 - Service call executed correctly
- $>0$ - Abnormal termination

This service call uses the ALGID specified to retrieve the private key associated with PRIKEYID and the user identification supplied by the host from the Cryptomodule's Database and uses this key to RSA decrypt the key associated with ENCRYPTEDKEY. It stores the decrypted key called KEYID in the Key Database and returns the resulting STATUS to the host.

# Appendix C

# Sample Implementation of rpc.rexd Client

**John Barkley**

```
/***************************************************************************/
/*                                                                         */
/*  A sample implementation of the "on" command (a client for rpc.rexd)    */
/*   which does not require the default directory on the client to be      */
/*                      exported to the server                             */
/*                                                                         */
/*         this implementation was developed under SunOS 4.1.1             */
/***************************************************************************/

#include <stdio.h>
#include <signal.h>
#include <rpc/rpc.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/time.h>
#include <netdb.h>
#include "rex.h"

main (argc, argv)
int argc;
char **argv;
{
extern char **environ;
extern int errno;
static rex_start rs ; /* rpc argument pointer */
static rex_result rr ; /* rpc result pointer */
static char nulstr[] = "";

struct timeval total_timeout;
int rpcsock = RPC_ANYSOCK;
register CLIENT *client;
enum clnt_stat clnt_stat;
```

```
/*           socket stuff                         */

int sockin, sockout, length;
struct sockaddr_in clientin, server;
int msgsockin, msgsockout;
static char buf[1024], *bp;
int inc, outc, wc;
int i, pid;

if (argc < 3)  {
fprintf (stderr, "usage: myon host command\n");
exit (-1);
}

rs.rst_cmd.rst_cmd_len = argc-2;
        rs.rst_cmd.rst_cmd_val = &argv[2];
rs.rst_host = &nulstr[0];
rs.rst_fsname = &nulstr[0];
rs.rst_dirwithin = &nulstr[0];
for (i=0; environ[i] != 0; i++)
rs.rst_env.rst_env_len = i;
rs.rst_env.rst_env_val = environ;
rs.rst_flags = 0;

/* Get a TCP CLIENT pointer */

if ((client = clnt_create(argv[1], REXPROG, REXVERS,
      "tcp")) == NULL)  {
clnt_pcreateerror ("clnttcp_create");
exit (-1);
}
```

```
/*      create input and output sockets for rpc.rexd to connect to      */

sockin = socket(AF_INET, SOCK_STREAM, 0);
if (sockin < 0 ) {perror("opening input stream socket"); exit(-1); };

sockout = socket(AF_INET, SOCK_STREAM, 0);
if (sockout < 0 ) {perror("opening output stream socket"); exit(-1); };

/*          name sockets            */
clientin.sin_family = AF_INET;
clientin.sin_addr.s_addr = INADDR_ANY;
clientin.sin_port = 0;
if (bind(sockout, (struct sockaddr *)&clientin, sizeof(clientin)) <0)
{perror("getting client socket name"); exit(-1); };

server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = 0;
if (bind(sockin, (struct sockaddr *)&server, sizeof(server)) <0)
{perror("getting server socket name"); exit(-1); };

/*         get assigned port numbers      */
length = sizeof(clientin);
if( getsockname(sockout, (struct sockaddr *)&clientin, &length) < 0)
{perror("getting client socket name"); exit(-1); };
length = sizeof(server);
if( getsockname(sockin, (struct sockaddr *)&server, &length) < 0)
{perror("getting server socket name"); exit(-1); };
rs.rst_port0 = ntohs(clientin.sin_port);
rs.rst_port1 = ntohs(server.sin_port);
rs.rst_port2 = rs.rst_port1;

/*              start accepting connections           */
listen(sockin, 5);
listen(sockout, 5);

/*                     Set UNIX style authentication                    */
/*                                                                      */
/*  to use DES authentication, replace with a call to authdes_create()  */

client->cl_auth = authunix_create_default();
```

```
/*              call rpc.rexd                          */

total_timeout.tv_sec = 20;
total_timeout.tv_usec = 0;
clnt_stat = clnt_call (client, REXPROC_START, xdr_rex_start, &rs,
                               xdr_rex_result, &rr, total_timeout);

if (clnt_stat != RPC_SUCCESS) {
clnt_perror (client, "rpc");
exit (-1);
}
if (rr.rlt_stat != 0){
fprintf(stderr,rr.rlt_message);
exit(-1);
}
/*   create parent and child processes for sockout and sockin communications */
pid = fork(); if(pid < 0){perror("fork"); exit(1); };
if (pid == 0)
/*              the child  for sockout                                  */
{
 close(sockin);
 msgsockout = accept(sockout, (struct sockaddr *)0, (int *)0);
 if( msgsockout == -1) {perror("accept sockout"); exit(1); };
 inc = 1;
 while (inc > 0)
 {
  errno = 0;
  inc = read(0, buf, sizeof(buf));
  if (inc < 0) perror("stdin read");
  if (inc <= 0) continue;
  bp = buf; outc = inc;
  while (outc > 0)
   {
    errno = 0;
    wc = write(msgsockout, bp, outc);
    if (wc <= 0) break;
    outc = outc - wc; bp = bp + wc;
    }
   }
 close(msgsockout);
 close(sockout);
 }
```

```
    else
/*                the parent for sockin                */
{
  close(sockout);
  msgsockin = accept(sockin, (struct sockaddr *)0, (int *)0);
  if( msgsockin == -1) {perror("accept sockin"); exit(1); };
  inc = 1;
  while(inc !=0 )
  {
   bzero(buf, sizeof(buf));
   if( (inc = read(msgsockin, buf, sizeof(buf))) < 0)
{perror("reading stream message"), exit(0); };
   if (inc != 0) write(1, buf, inc);
            }
  (void) kill(pid, SIGKILL);
 } ;
close(msgsockin);
close(sockin);
}
```