Harvard University Aiken Computation Laboratory technical report TR-05-87R

An Integrated Toolkit for Operating System Security

Michael O. Rabin and J. D. Tygar[1]

Aiken Computation Laboratory

Harvard University

Cambridge, MA 02138

August 1988

[1]The second author's current address is Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213. This research was supported in part under NSF research grant DCR-81-21431 at Harvard University. The second author received additional support from graduate fellowships from IBM and NSF.

Contents

Chapter 1

Introduction

Public and private organizations maintain large systems of files to be accessed by many users. Clearly, access to information in these files must be regulated so that specific items are made available to specific users, in accordance with rules and limitations deemed appropriate by management. The totality of these rules and limitations constitute the (information) *security policy* of the organization.

Nowadays such systems of files reside within computer systems, usually on secondary memory devices such as magnetic or optical disks or magnetic tapes. The files are accessed through computers, on terminals or printers, in a time-shared mode on a single computer, or in a distributed system of computers and file servers linked together. Computer systems are governed by operating systems which, among other tasks, implement and regulate the total user interaction with the system, including user access to the file system. Any hostile impingement on the integrity of the operating system poses grave dangers to the security of the file system, as well as to the proper intended behavior of the computer. Thus security requirements include the protection of operating systems from unauthorized incursion and subversion.

We feel that a method or *model* for implementing secure operating systems should posses the following attributes:

1. It should include mechanisms allowing the transaltion of any desired well specified security policy into the behavior of the given operating system.

2. It should provide software support and tools for facilitating the above process of translation.

3. It should ensure the proper intended behavior of the system even under malicious attacks.

4. Finally, despite its vital importance, computer security must be achieved at just a modest cost of system performance degradation.

Our approach to the problems of computer security, while bringing to bear some sophisticated algorithms, is at the same time pragmatic. We do not axiomatize the notions and mechanisms of security. Neither do we propose to conclusively demonstrate that a large software system possess certain security properties by carrying out formal verification. In fact, there are solid scientific reasons to believe that such global verification is not possible, and in practice formally verified secure system kernels have been found to have serious weaknesses [Benzel 84], [DeMillo-Lipton-Perlis 79], [Jelen 85], [McLean 85], [McLean 86], [Thompson 84].

What we rather do is to prepare a list of desired and essential properties that a computer system should possess so that a security policy can be reliably and conveniently implemented by users. The issue of security is viewed as that of controlling the access of users to files and of protecting the integrity of the operating system. This is effected through a series of new concepts, mechanisms, calculi, and software support constructs. Taken together these tools (the word is used in the colloquial sense rather than as in "software tools") comprise the *ITOSS (Integrated Toolkit for Operating System Security) model* for computer security. This model is general in scope and applicability and was also implemented in detail for the UNIX BSD 4.2 operating system.

Users act in the system through computing processes which make system calls requesting access to files. In our system, processes $\mathcal{P}$ have privileges V, and files $\mathcal{F}$ have protections T. We develop a formal calculus for representing privileges and protections by *security expressions*, and a semantics for the interpretation of such expressions as having values which are certain set constructs. We define the relation of a *privilege V satisfying a protection T* ($V \Rightarrow T$). By stipulation, a process $\mathcal{P}$ with privileges V will be allowed to access a file $\mathcal{F}$ with protection T if $V \Rightarrow T$. An efficient algorithm is developed for determining, for given expressions V and T, whether $V \Rightarrow T$.

Our treatment of privileges and protections includes *non-monotone* privileges, a construct of *indelible* protections; and a mechanism for enforcing *confinement* (see [DOD 85], [Lampson 73]).

The next tool of ITOSS is that of *incarnations*. From an organizational point of view, the significant entity is not the individual user as a person but the *role*, such as department head or bank teller, in which he interacts with the computer system. We therefore enable the organization to dynamically specify a set of entities $I_1, I_2, \ldots$ called *incarnations*, where each I_i is endowed with the privilege V_i deemed appropriate for the organizational role that I_i represents. The privilege V_i is passed to the processes created on behalf of I_i. A particular user (person) $\mathcal{U}$ will have specific incarnations $I, I', \ldots,$ associated with him. When $\mathcal{U}$ logs in he chooses the incarnation representing the role in which he intends to interact with the file system. By use of windows, he can simultaneously interact in several roles, without danger of security tresspasses.

The incarnations mechanism has several benefits. It allows precise tailoring of access privileges to needs of the work to be done by a user in a computer session. Access privileges can be specified according to work roles and can be effortlessly shifted around by system managers simply by removing or adding incarnations to a user.

Up to now we have described passive protection mechanisms. A *sentinel* S is a pointer to an executable file (program), call it F_S. The sentinel S may be incorporated as part of the protection header of a file $\mathcal{F}$. The code in F_S, and the decision whether to protect a particular file F by S, are made by system managers. The operating system includes code which, upon a system call to open a file $\mathcal{F}$, scans the header of $\mathcal{F}$ and creates sentinel processes $\mathcal{S}_1, \mathcal{S}_2, \ldots$, where $\mathcal{S}_i$ runs the code F_{S_i}, corresponding to the sentinels $S_1, S_2, \ldots$, listed in the header. Actually, the sentinels appear in the header in sentinel *clauses* of the form $C \rightarrow S$, where C is a *triggering condition.* The operating system tests C and invokes $\mathcal{S}$ as a process if and only if C is true.

Sentinels have many security and system applications. For example, we may write an audit program $F_{S_{\text{audit}}}$ which will record details of accesses to file $\mathcal{F}$ in a file $\bar{\mathcal{F}}$. For a sensitive file $\mathcal{F}$, one may include the sentinel S_{audit} in the protection. When such a file $\mathcal{F}$ is opened, the sentinel $\mathcal{S}_{\text{audit}}$ is invoked. The system passes to $\mathcal{S}_{\text{audit}}$ certain parameters which may include the file name of $\mathcal{F}$, an identifier of the process $\mathcal{P}$ which made the call to open $\mathcal{F}$, the name of the file $\bar{\mathcal{F}}$ into which the access to $\mathcal{F}$ by $\mathcal{P}$ will be recorded, etc.

It is important to emphasize that, as a rule, the decision to have any particular sentinel S, the code of F_S, the list parameters to be passed to $\mathcal{S}$, and the decision as to which file $\mathcal{F}$ will be protected by a clause $C \rightarrow S$ (and with which C), are all made by the system management. ITOSS provides the general sentinel *mechanism* as a *tool* which may then be employed by management for any purposes deemed useful.

In every computer installation there is a group of users, the system programmers, each of whom necessarily possesses extensive access privileges. Thus for UNIX, system programmers have "root", i.e. total, privileges. This poses very serious security threats. In ITOSS we have the *secure committee* tool. This mechanism allows management, if they so desire, to subject an incarnation I_{comm} to a committee of n users $\mathcal{U}_1, \ldots, \mathcal{U}_n$ (more accurately, n incarnations $I_1, \ldots, I_n$ of these users), a quorum of q of whom are required to invoke and later control I_{comm}.

Secure committees with any specified privileges are created by system management according to need, and the system can support any number of such committees. We shall explain later how to initiate this process so as to ensure security from the start. Secure committees have additional security applications beyond "watching the guards".

Additional security tools of ITOSS are *fences*. These include a method of "fingerprinting" system calls and files, and comparing fingerprints at appropriate points during the progress of the computation. A discrepancy between a pair of fingerprints which ought to be the same is an indication that some inadvertent or maliciously induced departure from the intended course of the computation has occurred. Fences are incorporated as modules into the kernel and run as part of the kernel code. They serve as a second line of defense against possible security weaknesses. Since it is possible to incorporate flexible variants of fences into operating system kernels *after* their completion, there is a good chance that consequences of any security error in a given kernel will be blocked at runtime by the fence.

In our implementation of ITOSS in conjunction with UNIX 4.2, we introduced fingerprinting at the top level of system calls, and again at the device driver level. This fence uncovered a hitherto unknown security breach arising from a possible race between `link()` and `chmod()` calls referring to a file. Even if the original code were left uncorrected, the unintended change of protection would be consistently blocked by the fence, and the attempted, unauthorized change of protetction of a file would be brought to our attention whenever arising during run time.

A centrally important feature of ITOSS is that when incorporated into an operating system it provides for more than just one option of a security policy. The tools of ITOSS allow the specification and implementation of a wide variety of organizational security policies.

Current proposals for high-security operating systems entail a serious degradation of system performance as a price for enhanced security. An important focus of our work is this issue of efficiency. The overall approach adopted by us, coupled with carefully chosen data structures and very fast algorithms for the frequently repeated security functions, results in a highly efficient system. In tests comparing our system with pure UNIX 4.2, we found no more than a 10% degradation of performance resulting from incorporating ITOSS.

Altogether, we feel that the tools developed in this work lead to the following security advantages:

1. ITOSS provides user-privilege and file-protection structures rich and fine-grained enough to faithfully express and implement any desired security policy. Furthermore, our formalism allows convenient and succinct expression of privileges and protections.

2. The coarseness of privilege/protection structures in existing security systems forces system designers, in certain instances, to confer excessive privileges of access on user and computing processes. These excessive privileges may then be employed by users and processes to subvert file and system security. The rich structure of privileges and protections of ITOSS, coupled with the tool of incarnations, allows tailoring of user and process privileges to the exact access requirements of the tasks to be performed. This circumvents many possible security pitfalls and dangers.

3. Sentinels provide convenient and reliable mechanisms for guarding against inadvertent or malicious failures of a user to follow security procedures, and for monitoring, auditing, and controlling access to sensitive files.

4. We provide system defenses against so-called "Trojan Horses" in application programs. These "Trojan Horses" are programs submitted by an unscrupulous party for general use within the system, and containing code which

when run by an unsuspecting user will illegally read, modify, or destroy his files. These defenses may be effected through the use of incarnations, fences, sentinels, non-monotonic protections, indelible protections and other ITOSS mechanisms.

5. Fences provide a "second line of defense", i.e. measures for detecting unauthorized changes of files and attempts to perform illegal accesses to files, reporting on the former and preventing the latter.

6. In existing computer systems there are always individual users, such as system programmers, with total access privileges. No system safeguards are provided against such pivotal individuals, should they wish to subvert the system's security. The mechanism of secure committees solves this problem.

7. The overhead cost in performance degradation resulting from running ITOSS is small (less than 10% slowdown over pure UNIX 4.2 operation.)

8. By implementing suitable changes, ITOSS can be incorporated into any existing operating system with a relatively modest programming effort, and will confer on that system the security protections of ITOSS.

Finally, a word on how ITOSS will be used. This system is intended for installations and organizations, such as banks, hospitals, corporations, and government departments, where security of information is an important issue. We envision that in such organizations, management will formulate a security policy, i.e. define organizational roles as well as classes of files (depending on security considerations) and specify, based on work needs, for every role, the set of files that can be accessed by users acting in that role.

The computer installation will have "security engineers" who are system analysts and programmers familiar with the tools and provisions of ITOSS, and also with the organization served by the system. These security experts will assist management in formulating the organization's security policy.

Actually we expect that with time and experience, security policies typical to various industries such as banking, retailing, and government, will emerge. Management of a particular organization will adopt and adapt a standard policy to its needs, without having to start from scratch.

The security engineers will implement the organization's security policy. This will include the creation of classes of privilege and protection expressions; the creation of incarnations; the assignment of privileges to incarnations and of protections to files. After proper system initialization, the assignment of protections to newly created files will as a rule be automated. The reasons for this mode of automatic assignment of protections to files, and the mechanism for doing it, are detailed in Chapter 4. Security engineers will decide which sentinel programs to choose from a standard library, and what new sentinels to create. They will create and install secure committees for performing extra sensitive tasks.

The day to day maintenance of security will also be in the hands of the security engineers (acting as a secure committee, if so prescribed by policy). They will initially introduce users into the system and assign incarnations to those users according to instructions from management. The security engineers, together with other system programmers (also acting as a secure committee), will be responsible for ongoing changes in the operating system, the file system modifications, and the updating of security features.

The features and mechanisms of ITOSS will be mostly transparent to the ordinary user. The user will be presented, after he logs in using his personal password, with a menu of the incarnations available to him. These will be stated in everyday terms such as: "Manager of Loans", "Member of Committee on Salaries", "Personal Matters." Once the user makes a selection of an incarnation, his access privileges and the protections for the files that he creates are automatically determined. He is oblivious to the details of privileges and protections, and to the existence of sentinels which may guard files that he accesses. The flexibility of ITOSS allows management to depart from the transparent-to-user mode, if so desired. For example, security policy may permit a user to set and modify protections, including sentinels, to his private files which he accesses in the Personal Matters role (incarnation) that may be available to him. All such details are questions of policy, and any kind of policy is implementable in ITOSS.

We feel that this division of labor and responsibilities for security between management, expert security engineers, and users, coupled with the tools and flexibly employable protections of ITOSS, will provide usable, reliable, and appropriate protection for information systems.

NOTE TO MICHAEL: WE NEED TO EXPLAIN INITIALIZATION

Chapter 2

Pure Access Control

2.1 Privileges and Protections

From management's point of view the issue of the security of information can be expressed as follows: We have a group of users and a dynamically changing body of information, which for the purposes of this work will be thought of as being organized in units called files. Management wants to define and enforce a regime specifying, for every user $\mathcal{U}$ and every file $\mathcal{F}$, whether $\mathcal{U}$ is allowed to access $\mathcal{F}$.

We view the security problem in the context of an operating system. In this environment the files reside in some kind of storage. Users have computing processes acting on their behalf.

Thus the security problem is reduced to being able to *specify* for every process $\mathcal{P}$ present in the system and every file $\mathcal{F}$ whether $\mathcal{P}$ will be allowed to access $\mathcal{F}$ and being able to *enforce* the specified regime. On the most general level, such a regime can be specified and enforced in one of the following three equivalent ways: We can create and maintain an *access matrix* M in which $M[i, j] = 1$ if and only if process $\mathcal{P}_i$ is allowed access to file $\mathcal{F}_j$. [Lampson 74] Alternatively, each process $\mathcal{P}_i$ may be provided with a *capability list* $L_i = (j_1, j_2, \ldots)$ so that $\mathcal{P}_i$ may access $\mathcal{F}_j$ if and only if j appears in L_i. The dual to this approach provides each file $\mathcal{F}_j$

with an *access control list* $L'_j = (i_1, i_2, \ldots)$ so that $\mathcal{P}_i$ may access $\mathcal{F}_j$ if and only if i appears in L'_j. When a process attempts access to a file, the operating system checks the access matrix, the capabilities list, or the access control list to see if this access should be permitted. The dynamically changing nature of the ensembles of processes and files and the large number of objects involved render such a regime difficult to specify and to update. Also, large capability lists for processes or long access control lists for files give rise to storage and runtime inefficiencies.

Our approach is to approximate these most general schemes by associating with every process $\mathcal{P}$, a list of *privileges* $\mathcal{V}$ called the *combined privileges* and with every file $\mathcal{F}$ a list of *protections* $\mathcal{T}$ called the *combined protections*. Access of $\mathcal{P}$ to $\mathcal{F}$ is allowed if $\mathcal{V}$ *satisfies* (is sufficient for overcoming) $\mathcal{T}$. We shall do this so as to satisfy the following criteria:

1. The privilege/protection structure must be sufficiently rich and fine grained to allow modelling of any access-control requirements arising in actual organizations and communities of users.

2. A formalism must be available so that users can rapidly and conveniently specify appropriate privileges for processes and protections for files.

3. There must be a rapid test whether a combined privilege $\mathcal{V}$ satisfies a combined protection $\mathcal{T}$.

When thinking about access of a process $\mathcal{P}$ to a file $\mathcal{F}$, we actually consider a number of access modes. For the purpose of this work we concentrate on the following modes:

1. Read

2. Write

3. Execute (i.e. run as a program)

4. Detect (i.e. detect its existence in a directory containing it)

5. Change Protection

This list of modes of accesses can be readily modified or extended to include other modes, according to need. Of these modes, Read, Execute, and Detect are henceforth designated as *non-modifying*, while Write and Change Protection are henceforth designated as *modifying.*

Both the combined privileges and the combined protection each consist of five privileges or protections, respectively, one for each of the access modes. Thus a process $\mathcal{P}$ has five privileges $(V_{\rm rd}, V_{\rm wr}, V_{\rm ex}, V_{\rm dt}, V_{\rm cp})$ associated with it, where $V_{\rm rd}$ is the read privilege, $V_{\rm wr}$ is the write privilege, etc. Similarly, a file $\mathcal{F}$ has five corresponding protections $(T_{\rm rd}, T_{\rm wr}, T_{\rm ex}, T_{\rm dt}, T_{\rm cp})$ associated with it. When process $\mathcal{P}$ makes a system call to read $\mathcal{F}$, the system will check whether $V_{\rm rd}$ satisfies $T_{\rm rd}$ before allowing the access. The other access modes are handled similarly. As we will see later, the detailed implementation is somewhat more complicated, and an check of a privilege may involve looking at several components of the protection. The reason for this complication is to provide for confinement (see section 2.6).

From now on we shall treat just a single privilege/protection pair $\langle V, T \rangle$, which can stand for any of the pairs $\langle V_{\rm rd}, T_{\rm rd} \rangle$, etc. In fact, $\langle V, T \rangle$ can stand for $\langle V_X, T_X \rangle$, where X is any additional access mode that an operating system designer may wish to single out.

As will be seen later, each privilege and protection have a finer detailed structure. Thus a privilege V_X, where X is a mode of access, will have several components.

2.2 The Tree of Securons

The basic atomic units out of which all privileges and protections are composed are nodes, called *securons*, of a specific tree. Since customers such as government departments or corporations, usually have tree-like hierarchical organizational structures, the tree of securons is a natural domain into which to map the desired security structure of the organization in question.

Definition 2.2.1 *The* securon tree *of* width n *and* depth h *is the set* $\mathbf{str}(n,h)$ *of all strings* $0.i_1.i_2.\cdots.i_k$ *where* $i_j \in [0 \; .. \; n-1]$*, and* $0 \leq k \leq h$*. The root node of this tree is* 0*.*

If $x, y \in \mathbf{str}(n,h)$ *and* $0 \leq i \leq n-1$*, then* y *is the* i*th* child *of* x *if* $y = x.i$*. The strings* x *and* y *are* related *if* x *is an initial sequence of* y *(denoted by* $x \leq y$*), or if* $y \leq x$*.*

The depth *of the securon* $x = 0.i_1.\cdots.i_k$ *is defined as* $d(x) = k$*.*

In every particular implementation of ITOSS a specific tree $\mathbf{str}(n,h)$ is used. In the current version of ITOSS, $n = 256$ and $h = 15$. We shall henceforth denote by **str** the fixed securon tree used in the architecture of our secure system.

As a first approximation we are tempted to use the securons themselves as privileges and protections. Thus we might assign to a manager the securon $x = 0.15.19.7$ and to his 15 subordinates the securons $0.15.19.7.i$, $0 \leq i \leq 14$. If a subordinate created a file $\mathcal{F}$, then his securon y would be attached as the protection $T = y$ of this file. A process $\mathcal{P}$ owned by the manager would have the privilege $V = x$. We could then stipulate that when the process $(\mathcal{P}, V)$ tries to access the file $(\mathcal{F}, T)$, permission would be granted only if $x \leq y$ (x is an initial sequence of y). We feel that such arrangements, and their obvious extensions and modifications, are not powerful and rich enough in complexity to reflect the security structure we need. However, in many existing systems the security structure is even more limited than in the above arrangement. We want to have much more sophisticated objects for expressing privileges and protections and a more flexible definition for the notion of a privilege satisfying a protection.

In particular, our security system will constrained layered constructs. The highest level structures are lists called called the *combined privileges (combined protec-*

tions). These have elements wheich are termed *privileges (protections)* which in turn consist of *privilege (protection) components.* Later we will refer to further subdivisions of the privilege (protection) components as *portions of the privilege (protection) components.* It is important to keep these layers distinct to fully understand the structure of ITOSS. The satisfaction relation is similarly defined in a layered manner; the satisfaction relation between combined privileges and combined protections for a particular mode of access will be defined in terms of a different satisfaction relation between a privilege and a protection which in turn is defined in turn of a third satisfaction relation between a privilege component and a protection component. Since these three satisfaction relations are defined over different domains, it will be clear which satisfaction relation we are referring to by the type of constructs being compared.

Definition 2.2.2 Privilege *components are sets $V \subseteq \mathbf{str}$ of securons and* protection *components T are sets of sets of securons (i.e. $T \subseteq \mathbf{P}(\mathbf{str})$, where $\mathbf{P}(S)$ denotes the set of all subsets of S).*

A privilege component

$$V = \{s_1, \ldots, s_m\}\,, \quad s_i \in \mathbf{str}$$

satisfies *(is sufficient to overcome) the protection component*

$$T = \{U_1, \ldots, U_m\}\,, \quad U_i \subseteq \mathbf{str}$$

if for some j, $1 \leq j \leq k$ *we have $U_j \subseteq V$. We shall use the notation*

$$V \Rightarrow T$$

to denote that V satisfies T.

These concepts allow us to realize in a simply manner very detailed access control list structures. Thus if we want to express the relation between manager and subordinates described above, we assign to the manager's processes the privilege containing the privilege component

$$V = \{0.15.19.7.i \mid 0 \leq i \leq 14\},$$

and to files of the ith subordinate, the protection $T_i = \{\{0.15.19.7.i\}\}$ (not $\{0.15.19.7.i\}$!) Now $V \Rightarrow T_i$ holds for all subordinates. If we wish to make subordinate 1's files accessible to an additional user $\mathcal{M}$, we can assign to $\mathcal{M}$ a securon s and define his set of securons to be $V_{\mathcal{M}} = H \cup \{s\}$, where H represents privileges required by $\mathcal{M}$ in other contexts. We also set $T_1' = T_1 \cup \{\{s\}\}$ and assign this protection component to the protection for each of the subordinate's files. Now we have $V \Rightarrow T_1'$ as well as $V_{\mathcal{M}} \Rightarrow T_1'$. We can, in fact, realize any access matrix by means of a sufficiently detailed assignment of privileges and protections.

2.3 Security Expressions

The above apparatus for implementing privileges and protections is indeed powerful but, as it stands, cumbersome. If privilege components $V \subseteq \mathbf{str}$ and protection components $T \subseteq \mathbf{P}(\mathbf{str})$ are to be specified by enumeration, then both the assignment of privileges and protections and the testing of whether $V \Rightarrow T$ will be too difficult to be of practical use. What we need is a formal calculus which will allow us to write quickly and economically compact formal expressions denoting rich and complicated privilege component and protection component sets. The first need is to describe large subsets of **str**. This will be done by introducing securon terms which will also be the atomic terms for building general expressions.

Definition 2.3.1 *Let* $s \in \mathbf{str}$ *and* $0 \leq i \leq j \leq \mathit{depth}(\mathbf{str})$. Securon terms *are:*

1. s

2. $s[i \ \mathbf{downto}\ j]$

Definition 2.3.2 *The set $SET(t)$ defined by a term t is $SET(s) = \{s\}$ for $s \in \mathbf{str}$, and*

$$SET(s[i \textbf{ downto } j]) = \{u \mid u \in \mathbf{str}, u \text{ related to } s, i \leq d(u) \leq j\}$$

Recall that u related to s means u is an ancestor, descendent, or is equal to s. Thus $SET(0[0 \textbf{ downto } \mathit{depth}(\mathbf{str})]) = \mathbf{str}$. And

$$SET(s[d(s) + 1 \textbf{ downto } d(s) + 1])$$

is the set of all children of the securon s.

Definition 2.3.3 *A* privilege expression *is defined by*

1. *Any securon term is a privilege expression.*

2. *If E_1 and E_2 are privilege expressions then so is $E_1 \wedge E_2$.*

Definition 2.3.4 *A* protection expression *is defined by*

1. *Any securon term is a protection expression.*

2. *If E_1 and E_2 are protection expressions then so are $E_1 \wedge E_2$ and $E_1 \vee E_2$.*

A *(security) expression* is a privilege or protection expression. Note that every privilege expression is a protection expression but not *vice versa.*

We must give semantics for these formal expressions, i.e., rules for associating a set $V \subseteq \mathbf{str}$ with a privilege expression, and for associating a set of sets $T \subseteq \mathbf{P}(\mathbf{str})$ with a protection expression. We shall use the notation $v_{\text{priv}}(E)$ to denote the value of the privilege component defined by the expression E, and $v_{\text{prot}}(E)$ to denote the value of the protection component defined by E.

If t is a securon term then we want to include in the corresponding privilege component all the securons in $SET(t)$. If $E = E_1 \wedge E_2$ is a privilege expression, we want the corresponding privilege component to be the weakest privilege component stronger than both the privilege component corresponding to E_1 and the privilege component corresponding to E_2. This motivates the following definition:

Definition 2.3.5 *The function $v_{\text{priv}}(E)$, giving the privilege component corresponding to the expression E, is defined by*

1. $v_{\text{priv}}(s) = \{s\}$, *for* $s \in \mathbf{str}$.

2. $v_{\text{priv}}(s[i\ \mathbf{downto}\ j]) = \{u \mid u \in \mathbf{str},\ i \leq d(u) \leq j\} = SET(s[i\ \mathbf{downto}\ j])$.

3. $v_{\text{priv}}(E_1 \wedge E_2) = v_{\text{priv}}(E_1) \cup v_{\text{priv}}(E_2)$.

When it comes to protections, we want the protection component corresponding to a term t to be the set containing *all singleton sets* $\{u\}$ for $u \in SET(t)$. We want the protection expression $E = E_1 \vee E_2$ to mean that any privilege component satisfying E_1 or satisfying E_2 also satisfies E. Finally, having the protection expression $E = E_1 \wedge E_2$ should mean that $V \Rightarrow E$ if and only if $V \Rightarrow E_1$ and $V \Rightarrow E_2$.

For sets (of sets) $T_1, T_2 \subseteq \mathbf{P}(\mathbf{str})$ we introduce the notation of cartesian product given by

$$T_1 \times T_2 = \{U_1 \cup U_2 \mid U_1 \in T_1,\ U_2 \in T_2\}$$

Notice that $T_1 \times T_2$ is again a set of subsets of $\mathbf{str}$.

Definition 2.3.6 *The function $v_{\text{prot}}(E)$ giving the protection component corresponding to the expression E is defined by*

1. $v_{\text{prot}}(s) = \{\{s\}\}$.

2. $v_{\text{prot}}(s[i\ \mathbf{downto}\ j]) = \{\{u\} \mid u \in SET(s[i\ \mathbf{downto}\ j])\}$.

3. $v_{\text{prot}}(E_1 \vee E_2) = v_{\text{prot}}(E_1) \cup v_{\text{prot}}(E_2)$.

4. $v_{\text{prot}}(E_1 \wedge E_2) = v_{\text{prot}}(E_1) \times v_{\text{prot}}(E_2)$.

If V is a privilege *expression* and T is a protection *expression* then V *satisfying* T ($V \Rightarrow T$) will of course be defined to mean $v_{\text{priv}}(V) \Rightarrow v_{\text{prot}}(T)$, where the latter relation was explained in Definition 2.2.2. From now on, we shall talk about privilege components and protection components as either expressions or sets, and the intended meaning, when not specified, will be clear from context.

2.4 An Algorithm for Testing $V \Rightarrow T$

The formalism of security expression developed in Section 2.3 allows us to specify, by writing concise expressions, large, complicated sets as privilege components and protection components. We need an efficient algorithm for testing whether a privilege components satisfies a protection component. Let $V = E_1 \wedge \ldots \wedge E_m$, where each E_i is a securon term, be a privilege expression and let T be a protection expression. Thus $T = G_1 \vee G_2$, or $T = G_1 \wedge G_2$, or T is a securon term. In the first case we test whether $V \Rightarrow G_1$, and if this fails we test whether $V \Rightarrow G_2$. In the second case we test whether $V \Rightarrow G_1$, if this fails then $V \not\Rightarrow T$, otherwise we continue to test whether $V \Rightarrow G_2$. Note that we adopt the so called *short-circuit evaluation* mode, where irrelevant branches are not pursued.

Thus, recursively, our problem is reduced to testing whether $V \Rightarrow t$ where t is a term, say, $t = s[i \textbf{ downto } j]$. The case $t = s$ is, trivially, a special instance of the former case. Let

$$V = s_1[i_1 \textbf{ downto } j_1] \wedge \ldots \wedge s_m[i_m \textbf{ downto } j_m].$$

Then $V \Rightarrow s[i \textbf{ downto } j]$ if and only if for some $k, 1 \leq k \leq m$,

$$SET(s_k[i_k \textbf{ downto } j_k) \cap SET(s[i \textbf{ downto } j]) \neq \emptyset. \tag{2.1}$$

To test (2.1) we introduce, for any $u \in \textbf{str}$ and integer $\ell \leq d(u)$, the notation $INIT(u, \ell)$ to denote the unique string v satisfying:

$$INIT(u,\ell) = \begin{cases} v & \text{where } v \leq u \text{ and } d(v) = \ell, \text{ if } \ell < d(u) \\ u & \text{if } d(u) \leq \ell \end{cases}$$

Denote $\bar{s} = INIT(s, i)$ and $\bar{s}_k = INIT(s_k, i_k)$. It is now readily seen that (2.1) is equivalent to the following easily checked condition:

$$(s_k \leq s \vee s \leq s_k \vee (\bar{s} \leq s_k \wedge \bar{s}_k \leq s)) \wedge ([i,j] \cap [i_k, j_k] \neq \emptyset). \tag{2.2}$$

Here $[i, j]$ denotes the interval of integers ℓ such that $i \leq \ell \leq j$.

A condition of the form (2.2) has to be tested for at most every term in T and every term in V. Hence

Theorem 2.4.1 *We can test whether $V \Rightarrow T$ holds in time $\mathit{length}(V) \cdot \mathit{length}(T)$.*

Let us indicate how to improve the algorithm. It follows from (2.2) that for $s_k[i_k \textbf{ downto } j_k] \Rightarrow s[i \textbf{ downto } j]$ to hold, $\bar{s}_k = \mathit{INIT}(s_k, i_k)$ and $\bar{s} = \mathit{INIT}(s, i)$ must be related (Definitions 2.2.1). By precomputing, we store the privilege component V in an array

$$\Big[\langle \bar{s}_1, s_1[i_1 \textbf{ downto } j_1]\rangle, \ldots, \langle \bar{s}_m, s_m[i_m \textbf{ downto } j_m]\rangle\Big]$$

so that for $i < j$,($\bar{s}_i \ \mathit{LEX} \ \bar{s}_j$), where *LEX* is a binary relation specifying the lexicographic ordering on strings (securons). To test whether $V \Rightarrow s[i \textbf{ downto } j]$ we use binary search, i.e., we check whether ($\bar{s}_k \ \mathit{LEX} \ \bar{s}$) for $k = \lceil m/2 \rceil$. Assume this holds, then we check whether $\bar{s}_k \leq \bar{s}$. If the latter does not hold, then $\bar{s}$ is not related to any $\bar{s}_\ell$, $1 \leq \ell \leq \lceil m/2 \rceil$, and the problem has been reduced by half. If $\bar{s}_k \leq \bar{s}$ does hold, then we check (2.2), and if that fails we continue binary search on *both* $[1, \lceil m/2 \rceil - 1]$ and $[\lceil m/2 \rceil + 1, m]$.

In practice, the combination of this binary search on V with the fact that the recursion on the structure of T usually does *not* lead to testing $V \Rightarrow t$ for every term t in T, results in running time about linear in $\log(\mathit{length}(V))$ and sublinear in $\mathit{length}(T)$. In actual experiments, the algorithm is very fast.

2.5 Negative Privileges and Protections

As formulated thus far, the power of privileges is monotonic, i.e., if V_1 and V_2 are privilege expressions and $v_{\text{priv}}(V_1) \subseteq v_{\text{priv}}(V_2)$ then whenever $V_1 \Rightarrow T$ it is also the case that $V_2 \Rightarrow T$. The planners of a secure file system may wish to make, in certain instances, access to some file $\mathcal{F}_1$ to be incompatible with access to file $\mathcal{F}_2$ because a user who possesses the combined information in $\mathcal{F}_1$ and $\mathcal{F}_2$ will have dangerous power.

To enable planners to implement such policies, and for other potential applications, we introduce the constructs of negative components of protections and negative components of privileges. The total effect will be to make the power of privileges *non-monotonic.*

Definition 2.5.1 Protections *have the structure* $T = \langle T_1, T_2 \rangle$ *where* T_1 *and* T_2 *are protection expressions. The first and second components of* T *are called, respectively, the* positive *and the* negative *protections of the file.*

Similarly, privileges will have the structure $V = \langle V_1, V_2 \rangle$*.*

A privilege $V = \langle V_1, V_2 \rangle$ satisfies $T = \langle T_1, T_2 \rangle$*, again written as* $V \Rightarrow T$*, if*

$$V_1 \Rightarrow T_1 \text{ and } V_2 \not\Rightarrow T_2.$$

When talking about a privilege $V = \langle V_1, V_2 \rangle$, we shall often use $V^+ = V_1$ and $V^- = V_2$ to denote the positive and negative components of V, and similarly for protections.

By way of illustration, if in the above example $\mathcal{F}_i$ was protected in the old sense by expression T_i, $i = 1, 2$, then protecting F_i by $\langle T_i, T_1 \wedge T_2 \rangle$, and using privileges of the form $\langle V, V \rangle$, will exactly enforce the desired discipline.

The second component of T can be left empty and then, by convention, $V \Rightarrow T$ if $V^+ \Rightarrow T^+$.

Remark: Definition 2.5.1 is preliminary and is intended to illustrate the effect of the positive and negative components in achieving non-monotinicity of privileges. Later on, when treating non-modifying privileges (see Definition 2.6.3 and the Change Protection privilege (Definition 2.7.1) we shall introduce additonal components into those privileges.

2.6 Indelible Protections and Confinement

Two additional enhancements of the privilege and protection structures are needed for addressing problems arising in file system security arrangements.

One of the modes of access to a file is the *Change Protection* mode which enables the user to change one or more of the file's protections. Since changing protections of a file may have far reaching and sometimes unforseen consequences, we want to have a mechanism for limiting the change of protection even when allowed.

To this end we introduce the notion of *indelible* security expression $!E$, where E is an expression. The intention is that the indelible portion of a protection component on a file cannot, as a rule, be changed by a process even if the process has Change Protection rights with respect to that file. It will, however, be seen later that a "bypass" exception to this preservation of indelible protections rule is needed.

Since the intention is to provide a floor below which a protection cannot be lowered, it is readily seen that incorporation of indelible components in a protection $\langle T^+, T^- \rangle$ should be in the manner

$$T = \langle T_1 \wedge !E, T_2 \vee !G \rangle. \tag{2.3}$$

In this way, any modification of T^+ leaves at *least* $!E$, and any modification of of T^- leaves at *most* $!G$.

We allow E and G to be written as any valid protection expressions. However, since E and G are indelible portions of protection components, we will give them a special interpretation. We will always treat E as a list of securons that are connected by $\wedge$ and G as a list of securons that are connected by $\vee$. NOTE: EXPLAIN WHY. The details of this interpretation are explained in the remainder of this section.

The concept of indelible protections is closely related to the issue of *confinement* of information. [Lampson 73] To illustrate confinement, assume that a file $\mathcal{F}$ has the Read protection $\langle !E, !G \rangle$, i.e., no mutable portion is present. Let another file

$\bar{\mathcal{F}}$ have the Read protection $\langle !\bar{E}, !\bar{G}\rangle$. Assume that some process $\mathcal{P}$ can read $\mathcal{F}$ and can *write* into $\bar{\mathcal{F}}$. The process $\mathcal{P}$ may then copy information from $\mathcal{F}$ into $\bar{\mathcal{F}}$. As matters now stand, there may be another process $\mathcal{P}_1$ which cannot read $\mathcal{F}$ but can read $\bar{\mathcal{F}}$. This process $\mathcal{P}_1$ may gain access to information in $\mathcal{F}$ via its copy placed in $\bar{\mathcal{F}}$ by $\mathcal{P}$. Our intention in providing indelible protections was to avoid unforseen declassification of information in a file through change of protection. It is reasonable to assume that for files guarded by indelible protections, we would also want to avoid inadvertent leakage of information in the manner described just now. To this end we introduce the concept of *confinement* and a mechanism to enforce it.

Define $\mathit{READERS}(\mathcal{F})$ by

$$\mathit{READERS}(\mathcal{F}) = \{\mathcal{P} \mid \mathcal{P} \text{ can read } \mathcal{F}, \text{ based on the indelible portions of the protections of } \mathcal{F}\}.$$

Similarly, define $\mathit{WRITERS}(\mathcal{F})$ by

$$\mathit{WRITERS}(\mathcal{F}) = \{\mathcal{P} \mid \mathcal{P} \text{ can write } \mathcal{F} \text{ based on the indelible portions of the protections of } \mathcal{F}\}.$$

Confinement can now be formally be defined as: *For any process $\mathcal{P}$ and files $\mathcal{F}, \bar{\mathcal{F}}$*

$$\mathcal{P} \in \mathit{READERS}(\mathcal{F}) \wedge \mathcal{P} \in \mathit{WRITERS}(\bar{\mathcal{F}}) \rightarrow \mathit{READERS}(\bar{\mathcal{F}}) \subseteq \mathit{READERS}(\mathcal{F}). \tag{2.4}$$

Let us emphasize that only the indelible portions of the protections of $\mathcal{F}$ and $\bar{\mathcal{F}}$ play a role in (2.4). Note that it also follows from the above definition of confinement that information can not leak from a protected file to an unprotected file by a *sequence* of reads, writes, and other accesses.

The above discussions lead to natural extensions of our previous privilege and protection structures. Our positive and negative protection components will (optionally) have indelible portions so that a typical protection with have the form

$$T = \langle T_1 \wedge !E, T_2 \vee !G\rangle \tag{2.5}$$

where T_1, T_2, E, and G are any protection expressions. By way of abbreviation, we shall use the notation $T = \langle T^+, T^- \rangle$ to describe (2.5) so that $T^+ = T_1 \wedge !E_1$ and $T_2 \vee !G$. We shall call T_1 and $!E$, respectively, the *mutable* and *indelible* portions of T^+, and similarly for T^-.

The semantics for $v_{\mathrm{prot}}(!H)$ will depend on the context, i.e. on whether $!H$ is part of T^+ or T^-. We accordingly introduce the notations v^+_{prot} and v^-_{prot} to mark this distinction.

To help simplify the specification of $v^+_{\mathrm{prot}}(!E)$ and $v^-_{\mathrm{prot}}(!G)$, we extend the notion $SET(t)$ introduced in Definition 2.3.2. for terms to the case of general security expressions H. Recall that $v_{\mathrm{prot}}(H) = \{U_1, \ldots, U_k\}$, $U_i \subseteq \mathbf{str}$.

Definition 2.6.1 *For any security expression H define*

$$SET(H) = \bigcup_{U \in v_{\mathrm{prot}}(H)} U.$$

Thus $SET(H)$ consists of all the securons "appearing" in $v_{\mathrm{prot}}(H)$. It follows that for a *privilege* expression E, $v_{\mathrm{priv}}(E) = SET(E)$.

Definition 2.6.2 *Let H be an expression. Define*

$$v^+_{\mathrm{prot}}(!H) = \{SET(H)\}.$$

$$v^-_{\mathrm{prot}}(!H) = \{\{s\} \mid s \in SET(H)\}.$$

$$v^+_{\mathrm{priv}}(!H) = v_{\mathrm{priv}}(!H) = SET(H).$$

The intention is that in $\langle T_1 \wedge !E, T_2 \vee !G \rangle$, $!E$ will give maximal positive protection and $!G$ will give maximal negative protection. Accordingly, we defined $v^+_{\mathrm{prot}}(!E)$ to be *all* of $SET(E)$ and $v^-_{\mathrm{prot}}(!G)$ to be *any* of $\{s\}$, $s \in SET(G)$.

Note that $v^+_{\mathrm{prot}}(!E)$ always has the form $\{\{s_1, s_2, \ldots\}\}$ and $v^-_{\mathrm{prot}}(!G)$ always has the form $\{\{s_1\}, \{s_2\}, \ldots\}$, where $s_1, s_2, \ldots$ are securons. For privileges, $v^+_{\mathrm{priv}}(!E)$ and $v^-_{\mathrm{priv}}(!G)$ are, as before, sets $\{s_1, s_2, \ldots\}$ of securons.

We now turn to the specification of privileges incorporating indelible expressions. Privileges for non-modifying actions (i.e. Read, Execute, and Detect) will have components whose functions will be to enforce confinement.

Definition 2.6.3 Privileges *for non-modifying modes of access have the form*

$$V = \langle V_1, V_2, M_1, M_2 \rangle \tag{2.6}$$

where V_1 and V_2 are privilege expressions, and M_1 and M_2 are any expressions. In (2.6), V_1 and V_2 are positive *and* negative *components of V, and M_1 and M_2 are called the* positive *and* negative mediating *components of V.*

We must also extend the notion of satisfaction for security expressions to the case where indelible expressions are included.

Definition 2.6.4 *Let V be a privilege component expression and $T^+ = T_1 \wedge !E$, $T^- = T_2 \vee !G$ be protection expressions. Define*

$$V \Rightarrow T^+ \ \text{ if } \ V \Rightarrow T_1 \ \text{ and } \ v_{\text{priv}}(V) \Rightarrow v^+_{\text{prot}}(E). \tag{2.7}$$

$$V \Rightarrow T^- \ \text{ if } \ V \Rightarrow T_2 \ \text{ or } \ v_{\text{priv}}(V) \Rightarrow v^-_{\text{prot}}(G). \tag{2.8}$$

In (2.7) and (2.8), the satisfaction relation on the right hand side is that of Definition 2.2.2, and its extension to expressions, explained after Definition 2.3.6.

Definition 2.6.5 *Let V_{rd}, with notation as in (2.6), and $T_{\text{rd}} = \langle T_1 \wedge !E, T_2 \vee !G \rangle$ be, respectively, a Read privilege and a Read protection. We shall say that V_{rd}* satisfies *T_{rd} ($V_{\text{rd}} \Rightarrow T_{\text{rd}}$) if*

1. *$V_1 \Rightarrow T^+$.*

2. *$V_2 \not\Rightarrow T^-$.*

3. *$SET(E) \subseteq SET(M_1)$.*

4. $\mathit{SET}(G) \subseteq \mathit{SET}(M_2)$.

The notation $V \Rightarrow T$ is similarly defined for the other non-modifying access privileges.

Definition 2.6.6 *Let $T = \langle T_1 \wedge !E, T_2 \vee !G \rangle$ and $\bar{T} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G} \rangle$, we shall say that $\bar{T}$* indelibly dominates *T if*

$$\mathit{SET}(E) \subseteq \mathit{SET}(\bar{E}), \quad \mathit{SET}(G) \subseteq (\bar{G}).$$

Lemma 2.6.1 *If $T_{\mathrm{rd}} = \langle T_1 \wedge !E, T_2 \vee !G \rangle$ and $\bar{T}_{\mathrm{rd}} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G} \rangle$ are Read protections for files $\mathcal{F}$ and $\bar{\mathcal{F}}$ respectively and $\bar{T}_{\mathrm{rd}}$ indelibly dominates T_{rd}, then it is the case that $\mathit{READERS}(\bar{\mathcal{F}}) \subseteq \mathit{READERS}(\mathcal{F})$.*

Proof: By definition, only the indelible portions of privileges play a role in determining $\mathit{READERS}(\mathcal{F})$ and $\mathit{READERS}(\bar{\mathcal{F}})$. Let $\mathcal{P}$ be a process with (indelible) Read privilege V_{rd} as in (2.6), such that $\mathcal{P} \in \mathit{READERS}(\bar{\mathcal{F}})$. Then, by definition,

$$v_{\mathrm{priv}}(V_1) \Rightarrow v^{+}_{\mathrm{prot}}(\bar{E}). \tag{2.9}$$

$$v_{\mathrm{priv}}(V_2) \not\Rightarrow v^{-}_{\mathrm{prot}}(\bar{G}). \tag{2.10}$$

$$\mathit{SET}(\bar{E}) \subseteq \mathit{SET}(M_1), \quad \mathit{SET}(\bar{G}) \subseteq \mathit{SET}(M_2). \tag{2.11}$$

Relation 2.9 is equivalent to $\mathit{SET}(\bar{E}) \subseteq \mathit{SET}(V_1)$ and relation 2.10 is equivalent to $\mathit{SET}(V_2) \cap \mathit{SET}(\bar{G}) = \emptyset$.

To say that $\bar{T}_{\mathrm{rd}}$ indelibly dominates T_{rd} means that $\mathit{SET}(E) \subseteq \mathit{SET}(\bar{E})$ and $\mathit{SET}(G) \subseteq \mathit{SET}(\bar{G})$. Hence the previous paragraph implies $\mathit{SET}(E) \subseteq \mathit{SET}(V_1)$ and $\mathit{SET}(V_2) \cap \mathit{SET}(G) = \emptyset$. Also, $\mathit{SET}(E) \subseteq \mathit{SET}(M_1)$ and $\mathit{SET}(G) \subseteq \mathit{SET}(M_2)$. Thus $V_{\mathrm{rd}} \Rightarrow T_{\mathrm{rd}}$ and $\mathcal{P} \in \mathit{READERS}(\mathcal{F})$. $\square$

Enforcing confinement means that if some process $\mathcal{P}$ reads $\mathcal{F}$ and can write into $\bar{\mathcal{F}}$, then $\mathit{READERS}(\bar{\mathcal{F}}) \subseteq \mathit{READERS}(\mathcal{F})$ should hold. This is insured by the following specification:

Definition 2.6.7 *Let a process $\mathcal{P}$ have the Read privilege $V_{\mathrm{rd}} = \langle V_1, V_2, M_1, M_2\rangle$ and Write privileges V_{wr}; and a file $\bar{\mathcal{F}}$ have the Read and Write protections $\bar{T}_{\mathrm{rd}} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G}\rangle$ and $\bar{T}_{\mathrm{wr}}$. The process $\mathcal{P}$ will be permitted to* write *$\bar{\mathcal{F}}$ if and only if*

1. *$V_{\mathrm{wr}} \Rightarrow \bar{T}_{\mathrm{wr}}$ (in these sense of Definition 2.5.1, disregarding the mediating components.)*

2. *$SET(M_1) \subseteq SET(\bar{E})$.*

3. *$SET(M_2) \subseteq SET(\bar{G})$.*

Theorem 2.6.1 *If for files $\mathcal{F}, \bar{\mathcal{F}}$ some process $\mathcal{P}$ can read $\mathcal{F}$ and write into $\bar{\mathcal{F}}$ then $READERS(\bar{\mathcal{F}}) \subseteq READERS(\mathcal{F})$. Thus confinement is enforced.*

Proof: Let $\mathcal{P}$ have privileges V_{rd} and V_{wr} as before and let $\mathcal{F}$ and $\bar{\mathcal{F}}$ have protections $T_{\mathrm{rd}} = \langle T_1 \wedge !E, T_2 \vee !G\rangle$, $\bar{T}_{\mathrm{rd}} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G}\rangle$, respectively. The conditions of the theorem and definitions 2.6.5 and 2.6.7 imply

$$SET(E) \subseteq SET(M_1) \subseteq SET(\bar{E}),$$

$$SET(G) \subseteq SET(M_2) \subseteq SET(\bar{G}).$$

Hence $\bar{\mathcal{F}}$ indelibly dominates $\mathcal{F}$, which by lemma 2.6.1 entails $READERS(\bar{\mathcal{F}}) \subseteq READERS(\mathcal{F})$. $\square$

2.7 Bypass Privileges

The introduction of indelible protections serves to enhance security by prohibiting inadvertent change of protections and by enforcing confinement. The intention in having indelible expressions is that they are unmodifiable even by a process that does have powerful Change Protection privileges V_{cp}. This strict interpretation of indelible protection creates practical difficulties. Clearly some carefully controlled processes must have the ability to change even indelible protections if such protections turn out to be too strict or if, later on, there arises the need for "declassifying" a file. Also, strict enforcement of confinement causes information to flow only upward in terms of security. Pragmatic needs require exceptions to this rule. The possibility to do so is realized through augmentation of the Change Protection Privilege.

Definition 2.7.1 *The* Change Protection privilege *has the structure*

$$V_{\text{cp}} = \langle V^+, V^-, B \rangle$$

where B is called the bypass component*. Let $\mathcal{F}$ be a file with Change Protection protection $T_{\text{cp}} = \langle T^+, T^- \rangle$ and $T_X = \langle T_1 \wedge !E, T_2 \vee !G \rangle$ be any of $\mathcal{F}$'s protections ($X = \text{cp}$ included). If $V \Rightarrow T^+$ and $V \not\Rightarrow T^-$ then the privilege V_{cp} is sufficient for changing the mutable components T_1 and T_2 in T_X to any other mutable expressions. If, in addition,*

$$SET(E) \subseteq SET(B) \text{ and } SET(G) \subset SET(B) \tag{2.12}$$

then the indelible components $!E$ and $!G$ can be changed as well.

Assume that a process $\mathcal{P}$ has Change Protection privilege V_{cp}, and a file $\mathcal{F}$ has Change Protection protection T_{cp} and another protection, say T_{rd}, with notations as above. If V_{cp} satisfies T_{cp} and (2.12) holds for T_{rd}, then $\mathcal{P}$ could read $\mathcal{F}$ by first changing T_{rd} to $\langle \{\emptyset\}, \emptyset \rangle$. Later $\mathcal{P}$ can restore T_{rd} to its original value $\langle T^+, T^- \rangle$. This motivates the following rule: *Under the above conditions, $\mathcal{P}$ can read $\mathcal{F}$; similarly, any other access X to $\mathcal{F}$ is permitted.*

2.8 Gradation of Actions on Files

Until now we made no assumption about the relative strengths of protections on a file $\mathcal{F}$. If we consider the effect of various modes of access to a file, we see that it does not make sense to have a weak Change Protection protection T_{cp} coupled with a strong Read protection T_{rd}. For a file protected in this way may be read by first changing the Read protection T_{rd} and then reading the file. Thus it makes sense to partially order the modes of access to a file and require that protections be correspondingly ordered.

Similar considerations apply to privileges associated with processes. Here the privilege for the more powerful action should be weaker. A reasonable, but by no means unique, gradation of modes of access to files is as follows. Changing protection of a file is the farthest reaching action, for by doing this all other modes of access can become available. Detecting the existence of a file $\mathcal{F}$ in a directory $\mathcal{D}$ is the minimal mode of access. For if a process, for example, writes into $\mathcal{F}$ it presumably knows of the existence of $\mathcal{F}$. Reading and executing a file are about equally powerful actions. For if a process $\mathcal{P}$ can read an executable file $\mathcal{F}$ then it can copy it to an executable file and subsequently execute it. Conversely, if $\mathcal{P}$ can execute $\mathcal{F}$ then by appropriate tracing the content of $\mathcal{F}$ can be fairly accurately reconstructed. Finally, there is no clear dominance between reading or writing into a file. These considerations lead to the semi-ordering of strengths of actions depicted in 2.8 above.

As explained before, protections on a file $\mathcal{F}$ should reflect this ordering. This does *not* mean, of course, that across different files $\mathcal{F}$ and $\bar{\mathcal{F}}$, the Change Protection protection $\bar{T}_{\mathrm{cp}}$ for $\bar{\mathcal{F}}$ should be stronger than, say, the Read protection T_{rd} for $\mathcal{F}$.

A convenient way for implementation order among protections is by syntactic means. We shall say that $\bar{T} = \langle \bar{T}_1 \wedge !\bar{E}, \bar{T}_2 \vee !\bar{G} \rangle$ *dominates* $T = \langle T_1 \wedge !E, T_2 \vee !G \rangle$ if for some expressions H_1, H_2,

$$\bar{T}_1 = T_1 \wedge H_1, \quad \bar{T}_2 = T_2 \vee H_2,$$

and, furthermore, $\bar{T}$ indelibly dominates T, i.e.

$$SET(E) \subseteq SET(\bar{E}), \quad SET(G) \subseteq SET(\bar{G}).$$

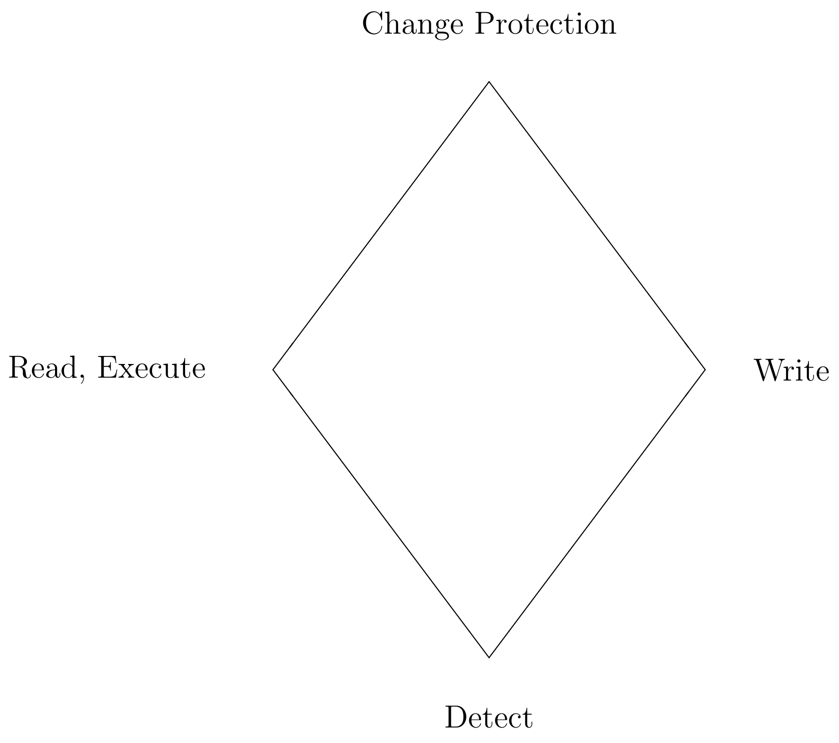

Figure 2.1: The hierarchy of access modes

Chapter 3

Sentinels

3.1 Overview

The mechanisms described in the previous chapter form a passive, *pure access scheme;* they describe when a process $\mathcal{P}$ can access a file $\mathcal{F}$ but the mechanisms perform no other action. While the pure access scheme gives us much power, there are basic security functions that it can not support. Thus, if we wanted to audit accesses to $\mathcal{F}$, that is record the names of all users who accessed a file $\mathcal{F}$, we would have to either modify the operating system kernel or modify all application code that might access $\mathcal{F}$. Since modifying and testing new code would be a laborious and dangerous process, this solution could only be rarely used and then with great difficulty. In this chapter we will extend the pure access scheme with *sentinels.* Sentinels allow us to conveniently add functions such as auditing. These functions can be customized to reflect any special needs an organization may have.

Here is how a sentinel for a file $\mathcal{F}$ works: Roughly speaking, a sentinel S is a name, listed in a $\mathcal{F}$'s protection header, of an executable file F_S. When any process $\mathcal{P}$ opens $\mathcal{F}$, the operating system schedules F_S for execution as a process $\mathcal{S}$. The process $\mathcal{S}$ is passed some parameters that allow it to perform various operations. The sentinel programs can be written to perform any desired action.

For example, if the security engineers wish to have an audit fucntion, they create a sentinel named, say, S_{audit} and write an appropriate program $F_{S_{\text{audit}}}$. The program $F_{S_{\text{audit}}}$ will accept as parameters the identity of the process $\mathcal{P}$ accessing a file $\mathcal{F}$, the name for $\mathcal{F}$ and for the file $\bar{\mathcal{F}}$ *into* which the access to $\mathcal{F}$ is recorded, and any other specified paremters. The code of $F_{S_{\text{audit}}}$ will realize the desired manner in which the access to $\mathcal{F}$ will be recorded in $\bar{\mathcal{F}}$.

Suppose we only wished to record access when a member of a certain class of users accessed $\mathcal{F}$. We could attach a sentinel S which checked whether the user accessing $\mathcal{F}$ belonged to the class. If he did, S would record the fact; if he did not, S would abort. While this is an adequate solution, it would be even better if we could keep overhead costs down by keeping process creation to a minimum. To allow this, we further extend the form in which a sentinel appears in a protection header to include a *trigger* condition R. Then sentinel S is only scheduled for execution by the operating system kernel when $\mathcal{F}$ is accessed *and* the process meets the trigger condition R.

Enhanced functionality is achieved by separating sentinels into different classes. Auditing simply requires that S make the necessary records in the audit file when $\mathcal{P}$ reads $\mathcal{F}$. A more sophisticated form of sentinels will take action only when certain records are read. For example, certain records in $\mathcal{F}$ might have a keyword `secret` attached to them. We may wish to have a sentinel S guarding $\mathcal{F}$ which will allow most accesses, but restrict access to `secret` records. The first example requires only an asynchronously running process, but the second example requires that S be able to continually monitor and approve individual I/O operations between $\mathcal{P}$ and $\mathcal{F}$. To achieve such mode of operation, we introduce two types of sentinel mechanisms: *asynchronous* sentinels which run independently of $\mathcal{P}$ and *synchronous sentinels* (also called *funnels*) which "lie between" $\mathcal{P}$ and $\mathcal{F}$. By "lie between", we mean that S is able to inspect and approve all I/O operations from $\mathcal{P}$ to $\mathcal{F}$. (In our implementation, funnels utilize UNIX *named pipes.* [Ritchie-Thompson 74], [Kernighan-Plauger 76], [Leffler-Fabry-Joy 83])

A very sophisticated attack might try to read a file $\mathcal{F}$ protected by an asynchronous sentinel S and then crash the operating system before S can perform its function. To prevent this, we further specialize sentinels to allow *Lazarus* processes which will, if interrupted by a system crash, be rerun when the operating system is rebooted.

When a sentinel S runs as a process, it must, as all processes, have some privileges. One choice an implementor could pick would be to have all sentinels run as the system user. This would not conform to the philosophy of this work which always calls for tailoring privileges to be the minimal ones necessary to perform the task at hand. In addition this would mean that we could not allow individual users to attach sentinels for their own purposes. To allow sentinels to be used in the widest possible context, we give each sentinel file F_S an assigned privilege V_S. When the sentinels is scheduled by the operating system, it will run with privilege V_S. A sentinel file is a special case of an exectuable file. In Chapter 4, we spell out the details of the assignment of privileges (more generally incarnations) to executable files and to sentinels.

It should be emphasized that we introduce sentinels as an operating system supported mechanism or tool. A particular version of ITOSS may come with a repertory of some standard sentinels which security engineers may use. But the sentinel tool allows security engineers to write any additonal sentinel files to perform useful security functions, and to routinely and effortlessly deploy sentinels as protections for files.

3.2 Semantics of Sentinels

Definition 3.2.1 *A* sentinel *S is an ordered tuple $\langle F_S, t\rangle$ where F_S is the name of a file, and t is the type,* 1, 2, *or* 3 *indicating that the sentinel S is*

1. Asynchronous*,*

2. Synchronous*,*

3. Lazarus*.*

Suppose a process $\mathcal{P}$ attempts to access a file $\mathcal{F}$. Suppose further that $\mathcal{P}$'s privilege for that access mode is V_P and $\mathcal{F}$'s protections for that access type are T including a sentinel S. First, the operating system tests whether $V_P \Rightarrow T$. This determines whether $\mathcal{P}$ can access $\mathcal{F}$. But regardless of the result, S can be executed as a sentinel. The operating system checks to make sure that F_S exists and is executable. If it is, $\mathcal{S}$ is created by the operating system with privileges V_S and executes according to type t. (See section 3.3 for a discussion of types of sentinels.)

To reduce unnecessary executions of F_S from being activated, we would like to express a trigger condition R specifying when S should be run by the operating system. There are several possibilities for expressing conditions R. We chose to use our privilege/protection scheme for this purpose, but it is easy to imagine other good choices for expressing trigger conditions.

Recall in equation (2.3) we saw the most general form of a protection,

$$T = \langle T_1 \wedge !E, T_2 \vee !G \rangle. \tag{3.1}$$

Definition 3.2.2 *A* sentinel clause C *is an ordered pair of a* trigger R *and sentinel* S*. The trigger* R *is a protection expression of the form of equation (3.1) and* S *is a sentinel as defined in Definition 3.2.1. We write* C *as* $R \rightarrow S$*.*

Suppose a process $\mathcal{P}$ with privileges V_P tries to access a file $\mathcal{F}$ protected by a sentinel clause $R \rightarrow S$, when $S = \langle F_S, t \rangle$. The operating system checks whether $V_P \Rightarrow R$. If it does, the operating system will run F_S.

We allow arbitrarily many sentinel clauses $C_1 = R_1 \rightarrow S_1$, $C_2 = R_2 \rightarrow S_2, \ldots$ to be attached to a protection. The operating system kernel will check each C_i, seeing whether $V_P \Rightarrow R_i$, and activating all the S_i which are triggered

Using this notation, we have a unified way of writing not just sentinel clauses, but also the pure access scheme. For example, if $\mathcal{F}$ has protection T and clauses $C_1, C_2, \ldots$ for access mode X, we can write an *extended protection for access mode X*,

$$(T \rightarrow ACCESS_X) \wedge (R_1 \rightarrow S_1) \wedge \ldots \wedge (R_k \rightarrow S_k) \tag{3.2}$$

where $ACCESS_X$ means that access mode X is permitted to the file.

There are two important exceptions to the above rules:

1. If $S = \langle F_S, t \rangle$ is a sentinel, then F_S can be an arbitrary executable file. In particular, F_S itself may be protected for Execute access by a sentinel S'. If S is triggered, it could lead to a chain of sentinels triggering sentinels — or even an infinite loop of sentinel scheduling. To prohibit this, we specify that a sentinel can only be triggered by a process $\mathcal{P}$ which is *not* a sentinel.

2. If a synchronous sentinel S guards a file $\mathcal{F}$ in its Change Protection protection, it can approve or disapprove each attempted action on $\mathcal{F}$. In particular, it could prevent *any* process from changing $\mathcal{F}$'s protection. Since the sentinel S is part of $\mathcal{F}$'s extended protection, S could never be removed. To prevent this, sentinels are not allowed to be attached to Change Protection protections.

3.3 Types of Sentinels

As indicated above, there are three types of sentinels: asynchronous, synchronous (also called funnels), and Lazarus. This section discusses technical issues related to the different types of sentinels. When process $\mathcal{P}$ triggers a sentinel $S = \langle F_S, t \rangle$

by accessing $\mathcal{F}$, the sentinel $\mathcal{S}$ is passed certain information through environment variables:

1. the name of $\mathcal{P}$,
2. $\mathcal{P}$'s privileges,
3. the name of $\mathcal{F}$,
4. $\mathcal{F}$'s protections,
5. parameters to the system call accessing the file,
6. other parameters which may be specified in the protection field of the $\mathcal{F}$ (these parameters are ignored by the protection mechanism and are used only by the sentinel receiving the data.) and
7. whether access was permitted by the pure access scheme.

Here are some technical details concerning our current implementation of synchronous sentinels in ITOSS. If the sentinel $\mathcal{S}$ is synchronous its input stream is triggered whenever $\mathcal{P}$ attempts to access $\mathcal{F}$. The data on the input stream to $\mathcal{S}$ contains the parameters of the I/O call to $\mathcal{F}$. While $\mathcal{S}$ executes, the process $\mathcal{P}$ is in the wait queue. The sentinel $\mathcal{S}$ can perform arbitrary actions on the input parameters. The sentinel $\mathcal{S}$ writes data to its output stream. That data forms the actual parameters used in the I/O call. If $\mathcal{S}$ writes a null field, the I/O call is refused. When the I/O call returns, the value is passed to $\mathcal{S}$. The sentinel $\mathcal{S}$ can then alter the return value. The sentinel $\mathcal{S}$ then is put in the wait queue while $\mathcal{P}$ continues. If several synchronous sentinels are triggered by one access, the I/O calls are passed to the sentinels in the same order they were triggered.

If S is Lazarus, a simple reliability mechanism is provided which protects sentinels from attacks which rely on crashing the system to avoid auditing. When S is triggered, a file system entry $\mathcal{E}$ is created in non-volatile memory before access to $\mathcal{F}$

is permitted. When $\mathcal{S}$ terminates normally, $\mathcal{E}$ is removed. However, if the operating system should crash or $\mathcal{S}$ is manually terminated, the entry $\mathcal{E}$ remains. When the system reboots, the boot procedure can detect $\mathcal{E}$ and retrigger S, activating F_S with the appropriate paremeters.[1]

3.4 Examples

We envision that a library of sentinels would provide basic security for most purposes. This library could be supplemented by any number of specially written sentinels for any desired security function.

Here are a few examples of how sentinels can be used. Some computer terminals have an *identify sequence.* This is a sequence of up to several hundred characters stored in random access memory within the terminal. The identify sequence is loaded from, and read back to, the central computer in response to command characters sent from the computer to the terminal. A *letter-bomb* is a piece of electronic mail containing character strings which first load a new identify sequence into the terminal, and then requests the identify sequence back. Thereby the sender of a letter-bomb can force any sequence of characters to be entered into the computer from a terminal. For example, the letter-bomb sender can force a sequence of characters to be sent which is a command deleting the terminal user's files. There are only two ways to protect against letter bombs in existing operating systems: either prohibit all terminals supporting identify sequences, or else put special code in the kernel (device driver) that prevents certain sequences of characters from being transmitted. Every time a new type of terminal is added, new code must also be added to the system. But sentinels yield a simple solution to the above problem: at-

[1]While the current version of ITOSS has not explicitly implemented Lazarus sentinels, they are available in the current implementation by using a two-phase commit protocol [Eswaran *et al* 76] in a synchronous sentinel and a special directory `/lock`.

tach a synchronous sentinel which checks for the dangerous sequences to the device files corresponding to the terminals which are vulnerable. The overhead incurred by including such a sentinel, if properly implemented, is very low, less than that of a window manager. In fact, this same sentinel could use other "forbidden sequences" (passed as paremeters) to prevent, if so desired, any particular piece of sensitive information containing a keyword, from being displayed on a particular terminal. This would be very useful if that terminal is located in an unsecured location.

Because sentinels are a powerful and general mechanism, we believe that they can be used in other software engineering applications outside of security.

NOTE: ELABORATE

3.5 Previous Work

The nearest previous concept to sentinels are *daemons*. (Introduced in the MULTICS operating system [Daley-Dennis 68] and the THE operating system [Dijkstra 68].) Daemons are continually running processes which maintain operating system functions such as providing for printing subsystems which require management of queues of files to be printed. Sentinels provide a much finer degree of control than daemons.

Several operating systems, such as MULTICS ([Organick 72], [Schroeder-Saltzer 72],), Hydra ([Wulf-Levin-Harbison 81), and UNIX ([Ritchie-Thompson 74]) attempted to address the issue of allowing a special process to actively intervene in I/O operations. The approach these systems adopted was to require that the intermediate processes — which correspond to our sentinels — be directly executed rather than passively triggered. Files were guarded by a restricted protection, and the intermediate process was granted powerful rights when executed. In UNIX, for example, each program $\mathcal{X}$ can have a set-UID bit set to be true. If it is, then $\mathcal{X}$ will assume the rights of the owner of $\mathcal{X}$ when it is executed. For example, a protected database would have a set-UID front-end; when data was changed, it could only be written by the front-end since no other ordinary user could write any bits of the database. Unfortunately, the only appropriate choice for most set-UID identities is the special user `root`, which can read or write any file in the system. Whenever `root` owns a set-UID program, that program becomes a potential source of security errors; since the set-UID program has arbitrary power in the operating system, the entire security structure of UNIX must be reproduced in every set-UID program. (Berkeley UNIX 4.2 has dozens of set-UID programs and there several widely known ways to violate the security of UNIX by exploiting weaknesses in those programs. Several methods that are also common to AT&T UNIX system V are described in [Grampp-Morris 84].)

Chapter 4

Incarnations and Secure Committees

4.1 Overview

A centrally important issue in secure operating and file systems is the correct and prudent management of privileges and protections, i.e., of access rights to files. There are several problems and dangers arising from the way access rights are assigned to *users* in existing systems.

A person often has a number of roles in which he is active within an organization, and which give rise to his interaction with the file system. In existing systems a user is usually presented to the operating system by a single entity through his login name, and this entity determines his access rights without regard to the purpose of his current computer session.

Assume that a user wishes to play a computer game. The program that he runs may have been inadequately tested and may contain Trojan Horse code. But the user's process running the program has access to *all* the files available to that user, so that the Trojan Horse code can cause serious damage.

Another difficulty arises when we have to add or revoke access rights. Assume that person $\mathcal{A}$ is in charge of a certain department and is chairman of a certain committee. These roles require access rights to two, possibly overlapping, sets of files. When person $\mathcal{A}$ is replaced by person $\mathcal{B}$ as chairman, the revocation of his access rights to the relevant files, and the assignment of these rights to $\mathcal{B}$ is a cumbersome and error prone process in existing systems.

In ITOSS the basic entities on whose behalf computing processes run are abstracted as *incarnations*. Each incarnation has combined privileges which will be attached to the processes created by it, and a scheme for attaching combined protections to the files created by these processes.

The set of incarnations is specified and dynamically updated by management, and generally reflects the organizational work roles. Each user has a number of associated incarnations with his login. When he logs in, he selects an appropriate incarnation from a menu of the incarnations available to him.

Consider the user $\mathcal{A}$ mentioned before. The security engineers create incarnations $\mathcal{I}_{\mathrm{head}\,X}$, $\mathcal{I}_{\mathrm{chmn}\,Y}$ to represent the roles of head of Department X and chairman of Committee Y, and an incarnation $\mathcal{I}_{\mathrm{min}}$ with minimal privileges. If user $\mathcal{A}$ wishes to run the game program, he invokes the incarnation $\mathcal{I}_{\mathrm{min}}$ which has no access rights to any significant files. If $\mathcal{A}$ is replaced by $\mathcal{B}$ as chairman of Committee Y, the security engineers remove $\mathcal{I}_{\mathrm{chmn}\,Y}$ from the incarnations available to $\mathcal{A}$ and associate it with $\mathcal{B}$.

A concern equally important to the aforementioned control of privileges of processes is the correct assignment of protections to files. ITOSS proves means for *automatic assignment* of headers, including combined extended protecteions, to files. This stands in contrast to most previous schemes which placed the brunt of responsibility for protection files on the individual user who created the file. In practice, the prevailing approach worked poorly: non-expert users who did not fully understand the working of the operating system would give incorrect security specifications. ITOSS, with its far richer ensemble of security mechanisms, would be an even greater challenge to non-expert users. Moreover, even well-informed users acting in an uncoordinated fashion could not properly manager the assignment of the global resource or individual securons.

Our innovation allws us to shift this responsibility to the security engineers. The protection assignment is performed by an operating system mechanism which takes into account the context in which a given file was created, insuring that filew will always be created with appropriate protections.

A prevalant difficulty with system security features is that they are generally unused by users of the computing system. The automated nature of ITOSS solves this problem.

In existing systems there are always some users, such as the system programmers, who have access to all of the system's resources and files. This arrangment poses obvious security dangers. In ITOSS, every role is represented by an incarnation. Usually an incarnation is assigned to a user who may in addition control several other incarnations, i.e., one user controls several incarnations. Dually, the *secure committee* tool allows management to subject control of any incarnation $\mathcal{I}$ to a *committee* of n users, so that a *quorum* of q committee members is required to invoke the incarnation $\mathcal{I}$ and execute commands through it.

Assume that the security engineers decide to create a secure committee SCOMM with certain access privileges, and wish to have SCOMM governed by a quorum of at least q of the users $\mathcal{U}_1, \ldots, \mathcal{U}_n$. They create a *committee incarnation* $\mathcal{I}_{\text{comm}}$ and *committee-member* incarnations $\mathcal{I}_1, \ldots, \mathcal{I}_n$ where $\mathcal{I}_j$ is assigned to user $\mathcal{U}_j$. If $k \geq q$ users $\mathcal{U}_{i_1}, \ldots, \mathcal{U}_{i_k}$ need to invoke $\mathcal{I}_{\text{comm}}$, then each $\mathcal{U}_{i_j}$ selects his committee-member incarnation $\mathcal{I}_{i_j}$. Every $\mathcal{U}_{i_j}$, $1 \leq j \leq k$ then makes through $\mathcal{I}_{i_j}$ a system call to invoke $\mathcal{I}_{rmcomm}$. These calls by $\mathcal{I}_{i_1}, \ldots, \mathcal{I}_{i_k}$ may involve, for greater security, "pieces" of a secret password in a manner explained in Section 4.3. The system creates $\mathcal{I}_{\text{comm}}$ only after having such $k \geq q$ calls from SCOMM members.

The operating system records the identifiers of the incarnations $\mathcal{I}_{i_1}, \ldots, \mathcal{I}_{i_k}$ (here

$q \leq k$) representing the users participating in the session. After the incarnation $\mathcal{I}_{\text{comm}}$ is invoked, every command by a joint shell owned by $\mathcal{I}_{\text{comm}}$ must be approved by all committee members participating in the session. Each participating member acts from his terminal and gets every system command for his inspection and approval.

The creation of secure committees, like all other security functions in ITOSS, is entrusted to the security engineers. As mentioned in the introduction, the security engineers and system programmers may also be organized into various secure committees, with possibly different quorums depending on the tasks and access privileges of the committee in question.

4.2 Incarnations

Essentially, in ITOSS, incarnations are the entities on whose behalf computing processes are running, and which control those processes. The technical details relating to incarnations are as follows. What is required is a specification of the combined privileges of processes $\mathcal{P}$ invoked by an incarnation, and a mechanism for assigning combined extended protections to the files created by such processes $\mathcal{P}$.

Definition 4.2.1 *An* incarnation $\mathcal{I}$ *has associated with it combined privileges* $(V_{\text{rd}}, V_{\text{wr}}, V_{\text{ex}}, V_{\text{dt}}, V_{\text{cp}}$ *and an* automatic assignment protection schema π*. (The structure of* π *is given in definition 4.2.2.)*

We allow each user $\mathcal{U}$ to have a number of incarnations $\mathcal{I}_1, \mathcal{I}_2, \ldots$. When $\mathcal{U}$ logs onto the computer, he must pick an incarnation. He can change his incarnation at any time. (On display terminals supporting windows, the incarnation can be displayed at all times to remind the user of which mode he is in.) All processes created by the user will have the default privileges specified by the incarnation. And all files created by such processes will have protections assigned to them according to the protection schema π of that incarnation. An important extension of this rule, in the case of executable files, is explained next.

Assume that a user $\mathcal{U}$ invoking incarnation $\mathcal{I}$ runs a process $\mathcal{P}$ executing program A. At a certain point $\mathcal{P}$ creates a new file $\mathcal{F}$. We wish to make the combined extended protection V attached to $\mathcal{F}$ dpend on: (a) the incarnation $\mathcal{I}$, i.e. the organizational role creating $\mathcal{F}$; (b) the program A; and (c) the type t of the file $\mathcal{F}$ (e.g. executable file, directory sensitive data, encrypted data, etc.), this type being known to the program A.

Definition 4.2.2 *Let* $\mathbf{C}$ *be a set containing names of library programs and the keyword* **other***,* $\mathbf{Z}$ *be the set of integers, and* $\mathbf{H}$ *be the set of combined extended protections. An* automatic protection assignment schema *is a function* $\pi : (\mathbf{C}, \mathbf{Z}) \rightarrow \mathbf{H}$.

The operational meaning of Definition 4.2.2 follows. Suppose $\mathcal{U}$ is in incarnation $\mathcal{I}$ with automatic protection assignment schema π. When $\mathcal{U}$ creates a new file $\mathcal{F}$ while using application program $\mathcal{A}$, the ITOSS protection assigner checks whether $\mathcal{A} \in \mathbf{C}$. If this is true, $\mathcal{A}$ is used as the first argument to π. If not, the keyword **other** is used. The program $\mathcal{A}$ can pass information about the type t requirements of the file as an integer through the second argument to π. second argument in $\mathbf{Z}$. If no value is specified, we set i to be zero. The file $\mathcal{F}$ is given the protection header $\pi(\mathcal{A}, i)$ or $\pi(\mathbf{other}, i)$ suggested by $\mathcal{I}$ if changing protection, as the case may be.

Furthermore, the semantics of the automatic protection schema π is extended to enable a user to safely change the protection of an existing file. If a user while in an incarnation $\mathcal{I}$ has change protections rights to a file $\mathcal{F}$, he could execute a command specifing a program name A and an integer i which would assign to the file $\mathcal{F}$ the combined protections that would have been assigned to $\mathcal{F}$ if it had been created by A with the integer i passed as the second parameter to π.

4.3 Secure Committees

In many organizations actions with major consequences are made by groups of people. For example, payments of large amounts of money may require co-signers. This eliminates risk arising from the corruptibility of a single person.

We wish to emulate this feature for system management. We define a *secure committee* to be an incarnation $\mathcal{I}_{\text{comm}}$ requiring a quorum q people out of a set of users $\{\mathcal{U}_1, \ldots, \mathcal{U}_m\}$ to approve all actions. As usual in ITOSS, the users $\mathcal{U}_1, \ldots, \mathcal{U}_m$ are represented in the system by incarnations $\mathcal{I}_1, \ldots, \mathcal{I}_m$. A *committee session* is an active meeting of at least q committee members, who are called the *participants*.

The incarnation $\mathcal{I}_{\text{comm}}$ can be invoked only through cooperation of q committee members. During a committee session, every proposed command will be presented to all of the participants. The command will be performed only if each of the participants explicitly approves it. Hence each participating committee member has veto power during the session in which he is active.

To implement secure committees, we need to use a secret sharing algorithm due to Adi Shamir. [Shamir 79] The algorithm gives a protocol for dividing a text t into m encrypted pieces such that q pieces determine the value of t, but $q-1$ pieces give *no* information about the value of t.

We do all computations with integer residues modulo a prime p. The value of p is chosen to be larger than m and than all possible values of t. Let

$$h(x) = r_1 x^{q-1} + r_2 x^{q-2} + \cdots + r_{q-1} x + t \qquad (4.1)$$

be a polynomial, where the r_i are independent randomly chosen integer residues. The pieces of the secret are the values $h(1), \ldots, h(m)$. The secret is $t = h(0)$. Since q values of h determine a nonsingular linear system of q equations in q variables, Lagrangian interpolation will allow us to recover the values of the coefficients of h, and hence t. On the other hand, $q-1$ values of h do not provide any information about the value of t.

For implementing secure committees, we let t be the password to the secure committee incarnation. A sentinel called the *dealer box* picks a random polynomial h of the form in (4.1), and securely distributes the pieces $h(1), \ldots, h(m)$ of the secret to appropriate files accessible by the incarnations $\mathcal{I}_1, \ldots \mathcal{I}_m$ of the individual committee members $\mathcal{U}_1, \ldots, \mathcal{U}_m$. (Secure distribution may use file system protection primitives or may depend on private key encryption.)

When a quorum of q users wish to form a secure committee, they pass their pieces of the password t to the dealer box which assembles the pieces of the secret and passes the value t to the password checking program. If the password checker accepts t, a secure committee incarnation $\mathcal{I}_{\text{comm}}$ is invoked. Meanwhile, the dealer box picks a new random password t' and random polynomial h'. After registering the password t' with the password checker, the dealer box securely distributes the new pieces of the secret $h'(1), \ldots, h'(m)$ to the committee members. (If the system should crash prior to completing distribution, the password checker will still accept t.) This keeps an opponent from slowly gathering pieces of t and being able to form a secure committee by himself.

The secure committee concept can be extended in several straightforward ways. The protocol can support "weighted quorums" where different numbers of members are required to form a secure committee depending on rank. For example, we can give each junior member one piece of the password and each senior member two pieces. The secure committee tool is also very appropriate for distributed implementations of secure operating systems.

If there are operations which do not require the full strength of the committee but are still sensitive, the committee can delegate these tasks to *subcommittees* by using sentinels. In the degenerate case, some simple sensitive tasks may be entrusted to a single user — a machine operator, for example.

Chapter 5

Fences

5.1 Validation

Once an implementor has specified a language for expressing security constraints and provided a mechanism for enforcing them, the task he faces is to *validate* the resulting system so to show that it is free of errors. Validation is an important issue not just for security but for software engineering in general, and a large number of methods, such as formal verification, testing, structured walkthroughs, have been proposed for dealing with this problem. In practice, none of these methods guarantee software without errors; they merely increase the confidence a user has in the system.

Validation for security is special because in many cases we are trying to prohibit some event from occurring. In this chapter, we propose a general method, *fences*, for providing a "second test" of security conditions.

The term "fence" was first applied to the IBM 7090 computer to describe a memory protection mechanism. [Bashe *et al* 86] In this context, a fence was a pointer into memory which separated user and system memory. Memory beyond the fence was accessible only in system mode, and this was enforced by independent hardware.

In our usage, a fence is any low-overhead hardware or software feature which enforces security conditions by testing values independently of the main stream of execution, allowing operations to be performed only if they do not violate security conditions.

5.2 Fingerprints

In the course of his research on string matching, the first author proposed a special hash function, called a *fingerprint*. [Rabin 81] His fingerprint function $F_K(x)$ hashes a n-bit value x into a m-bit value $(n > m)$ randomly, based on a secret key K. The interesting point is that given a y, if K is unknown, then no one can find an x such that $F_K(x) = y$ with probability better than 2^{-m}.

(Briefly, Rabin's algorithm picks an irreducible polynomial p of degree m over the integers modulo 2. The coefficients of p, taken as a vector, form the key K. The bits in the input x are taken as the coefficients of a $n-1$ degree polynomial q. Let r be the residue of q divided by p in $Z_2[x]$. r is a $m-1$ degree polynomial, and its coefficients, taken as a vector, form $F_K(x)$. A software implementation of this algorithm merely consists of a sequence of very fast XOR operations. [Rabin 81] gives this algorithm in greater detail. [Fisher-Kung 84] describes a very fast systolic hardware implementation of this algorithm.)

With the fingerprinting algorithm, we can install powerful fences. Suppose we wish to guarantee that a file $\mathcal{F}$ has not been tampered with. One way we could protect against this is by installing a synchronous sentinel to guard $\mathcal{F}$. However, $\mathcal{F}$ would still be vulnerable to attacks on the physical disk. As a second-tier protection, we could have the synchronous sentinel guarding $\mathcal{F}$ keep an independent fingerprint of $\mathcal{F}$ elsewhere in the operating system. If $\mathcal{F}$ was changed illicitly, the sentinel would instantly detect it unless the fingerprint was also changed. Since the fingerprint is provable impossible to forge with accuracy greater than 2^{-m} unless the key K is known, it is impossible for the opponent to change $\mathcal{F}$ without eventually coming to the attention of the sentinel.

5.3 System Call Fingerprints

A wide class of existing bugs in Berkeley UNIX 4.2 are based on race conditions. In UNIX, system calls such as "read file", "write file", and "change protection" are not atomic but concurrent. Because of the way the UNIX kernel is structured, it is very difficult to detect all possible race conditions.

One race condition exists between the `link` system call and the `chmod` system call. `chmod` changes the protection on a file $\mathcal{F}$. In UNIX 4.2, security is enforced by only allowing the owner of $\mathcal{F}$ or the system-user `root` to access $\mathcal{F}$, so it first checks ownership information and then changes the protection. `link` makes a new file system entry $\mathcal{F}'$ and then sets it to point to $\mathcal{F}$. Since a `link` call merely establishes a link, no special security rights are required to execute it.

By running the two system calls simultaneously, it is possible for an opponent to gain access to a file he does not own. Let $\mathcal{F}$ be owned by $\mathcal{U}$. An opponent $\mathcal{U}'$ might gain access to $\mathcal{F}$ by executing these two system calls simultaneously:

1. `link(`$\mathcal{F}'$`,`$\mathcal{F}$`)`. Link file $\mathcal{F}$ to $\mathcal{F}'$.

2. `chmod(0666,`$\mathcal{F}'$`)`. Make $\mathcal{F}'$ publicly available for reading and writing.

If these system calls are executed, the following chain of events sometimes will occur:

1. `link` creates a dummy entry $\mathcal{F}'$ owned (temporarily) by $\mathcal{U}'$.

2. `chmod` checks $\mathcal{F}'$ to see whether $\mathcal{U}'$ is allowed to change its protection. Since $\mathcal{U}'$ temporarily owns $\mathcal{F}'$, `chmod` approves the action.

3. `link` completes the pointer from $\mathcal{F}'$ to $\mathcal{F}$. At this point $\mathcal{U}'$ can not access $\mathcal{F}$ or $\mathcal{F}'$ since those files are owned by $\mathcal{U}$.

4. `chmod`, having already approved rights to change the protections to $\mathcal{F}'$, goes ahead and makes that file publicly readable and writeable. Since $\mathcal{F}'$ is now a pointer to $\mathcal{F}$, this makes $\mathcal{F}$ publicly readable and writeable, and thus $\mathcal{U}'$ can access that file.

We found this bug and several other bugs in UNIX 4.2 by inserting fences in the operating system which fingerprint system call requests at the top level of the kernel and at the driver level. The fingerprints contain the text of the request and the security data (process privilege information and file protection information) associated with the processes and files. If the fingerprints do not match, the fence determines that race conditions must exist in the kernel and halts the processor. In the above example, the protection information associated with $\mathcal{F}'$ changes from the top level of the kernel and the driver level and hence the fingerprints are different.

While our current library of fences is not yet sufficient to validate a secure system by itself, we believe that the technique can be used in conjunction with more traditional validation methods to provide a very high degree of confidence in security software.

Chapter 6

Bibliography

[Bashe 86] Bashe, C. J., L. R. Johnson, J. H. Palmer, and E. W. Pugh. *IBM's Early Computers* MIT Press, Cambridge, Massachusetts, 1986.

[Benzel 84] "Analysis of a Kernel Verification." *Proceedings of the 1984 Symposium on Security and Privacy*, Oakland, California, May 1984, pp. 125–131.

[Daley-Dennis 68] Daley, R. C., and Dennis, J. B. "Virtual Memory, Processes, and Sharing in MULTICS." *Communications of the ACM,* **11:**5, pp. 306–312 (May 1968).

[DeMillo-Lipton-Perlis 79] DeMillo, R. A., R. J. Lipton, and A. J. Perlis. "Social Processes and Proofs of Theorems and Programs." *Communications of the ACM,* **22:**5, (May 1979).

[Dijkstra 68] Dijkstra, E. W. "The Structure of the 'THE' Multiprogramming System." *Communications of the ACM,* **11:**5, pp. 341–346 (May 1968).

[DOD 85] *Trusted Computer System Evaluation Criteria.* Computer Security Center, Department of Defense, Fort Meade, Maryland. (CSC-STD-001-83) March 1985.

[Grampp-Morris 84] Grampp, F. T., and R. H. Morris. "UNIX Operating System Security." *AT&T Bell Laboratories Technical Journal,* **63:**8b, pp. 1649–1672 (October 1984).

[Jelen 85] Jelen, G. F. *Information Security: An Elusive Goal.* Program on Information Resources Policy, Harvard University, Cambridge, Massachusetts. June 1985.

[Lampson 73] Lampson, B. W. "A Note on the Confinement Problem." *Communications of the ACM,* **16:**10, pp. 613–615 (October 1973).

[Lampson 74] Lampson, B. W. "Protection." *ACM Operating Systems Review,* **19**:5, pp. 13–24 (December 1985).]

[McLean 85] McLean, J. "A Comment on the 'Basic Security Theorem' of Bell and LaPadula." *Information Processing Letters,* **20:**3, pp. 67–70 (1985).

[McLean 86] McLean, J. "Reasoning About Security Models." Personal Communication, 1986.

[Organick 72] Organick, E. I. *The Multics System.* MIT Press, Cambridge, Massachusetts, 1972.

[Rabin 81] Rabin, M. O. "Fingerprinting by Random Polynomials." TR-15-81. Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts. 1981.

[Ritchie-Thompson 74] Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System." *Communications of the ACM,* **17:**7, pp. 365–375 (July 1974).

[Schroeder-Saltzer 72] Schroeder, M. D., and J. H. Saltzer. "A Hardware Architecture for Implementing Protection Rings." *Communications of the ACM,* **15:**3, pp. 157–170 (March 1972).

[Shamir 79] Shamir, A. “How to Share a Secret.” *Communications of the ACM,* **22:**11, pp. 612–613 (November 1979).

[Thompson 84] Thompson, K. “Reflections on Trusting Trust.” *Communications of the ACM,* **27:**8, pp. 761–763 (August 1984).

[Wulf-Levin-Harbison 81] Wulf, W. A., R. Levin, S. P. Harbison. *HYDRA/C.mmp.* McGraw-Hill, New York, NY, 1981.