

2003

Domain and type enforcement in Linux

Serge Edward Hallyn

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Hallyn, Serge Edward, "Domain and type enforcement in Linux" (2003). *Dissertations, Theses, and Masters Projects*. Paper 1539623428.

<https://dx.doi.org/doi:10.21220/s2-0x9t-ag80>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Domain and Type Enforcement in Linux

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

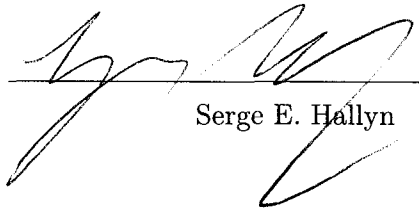
Serge Edward Hallyn

2003

APPROVAL SHEET


This dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy




Serge E. Hallyn

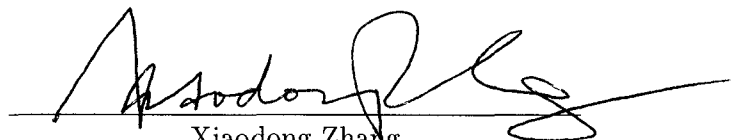
Approved, September 2003



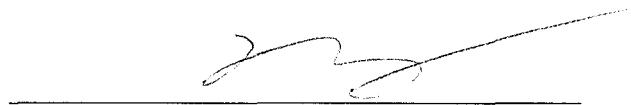
Phil Kearns
Thesis Advisor




Weizhen Mao



Xiaodong Zhang



Haining Wang



Jean Mayo
Michigan Technological University

To my mother.

Table of Contents

Acknowledgments	xi
List of Figures	xiv
Abstract	xv
1 Introduction	2
1.1 Contributions	2
1.2 Organization	3
2 Background	5
2.1 Security Nomenclature	5
2.2 Access Control Policies	6
2.2.1 Bell-La Padula (BLP)	6
2.2.2 Rings	8
2.2.3 Clark-Wilson	10
2.2.4 Strict Integrity	11
2.2.5 Type Enforcement	11

2.3	Policy Representation	13
2.3.1	Access Control Matrix	13
2.3.2	Access Control List	13
2.3.3	Capabilities	14
2.4	Role Based Access Control	16
2.5	UNIX	17
2.5.1	File System	17
2.5.2	File Access	19
2.5.3	Signal Access	20
2.5.4	Superuser	21
2.5.5	POSIX Capabilities	21
2.5.6	Domain and Type Enforcement	23
2.6	Linux File System Architecture	24
2.7	Stackable File System	26
2.8	Networking	28
2.9	Other Projects	29
2.9.1	Linux-ACL	29
2.9.2	LIDS	30
2.9.3	TE and DTE	31
2.9.4	SELinux	31
2.9.5	HP-LX	32
2.10	Other Work in Security Policies	33

3	DTE	34
3.1	LSM	35
3.1.1	LSM Design	36
3.2	DTE Design Decisions	38
3.2.1	Entry Types	38
3.2.2	File Type Resolution	40
3.2.3	Extended Attributes	42
3.2.4	Policy Updates	44
3.2.5	Networking	45
3.3	Data Structures	46
3.4	Algorithms	49
3.4.1	Mount	49
3.4.2	File Type Resolution	53
3.4.3	Inode Permission	56
3.4.4	Execve	57
3.4.5	DTE Module Init	60
3.5	Configuration File	61
3.6	DTE API	63
3.7	Effectiveness	65
4	Performance	68
4.1	LMBench Results	69
4.1.1	File System and VM Performance	69

4.1.1.1	Mmap	69
4.1.1.2	File Creation	70
4.1.1.3	File Deletion	70
4.1.2	Process-Related Performance	71
4.1.2.1	Null Call	71
4.1.2.2	Stat() and Open()/Close()	72
4.1.2.3	Signals	72
4.1.2.4	Fork	73
4.1.2.5	Fork and Exec	73
4.2	Micro Benchmarks	74
4.2.1	Permission()	74
4.2.2	Execve()	77
4.2.3	Signal	78
4.3	Macro Benchmark	79
5	Access Rights of Domains	80
6	Policy Administration Tools	87
6.1	DTEedit	93
6.2	DTEview	96
7	Analysis of DTE Policies	107
7.1	Using the BLP *-Property	108
7.2	Limitations of BLP	110

7.3	Modified BLP	112
7.3.1	MBLP Enforces Clark-Wilson CDIs	115
7.3.2	MBLP Enforces Assured Pipelines	116
7.4	Examples of Analysis Using MBLP	118
8	Construction of DTE Policies from Modules	123
8.1	Policy Compiler File Formats	125
8.1.1	Control File Specification	125
8.1.2	Module File Specification	128
8.1.2.1	Priority of Access Rules	132
8.1.2.2	Group Expansion	133
8.1.2.3	Inheritance	134
8.1.3	Patch File Specification	135
8.2	Automatic Maintenance of Policy Constraints	137
8.2.1	Correctness of the BLP PCP	138
8.2.2	Modified BLP PCP	142
8.3	Sample Modules	146
8.3.1	Base Module	146
8.3.2	Password Module	147
8.3.3	Ftp Module	147
8.3.4	Syslog	148
8.3.5	Control File	148
8.4	Conclusion	149

9	Implementation Extensions	150
9.1	Namespaces	150
9.1.1	Problem	151
9.1.2	Solution	152
9.2	Accommodating proc	153
9.3	Providing Network Security	157
9.3.1	Security	157
9.3.2	Convenience	161
9.3.2.1	Static Type Assignment	161
9.3.2.2	Server-Directed Type Assignment	162
9.4	Conclusion	164
10	Conclusion	169
A	DTE Policy, Modules, and PCP Listings	173
A.1	Sample DTE Policies	173
A.1.1	Base Policy	173
A.1.2	Password Policy	175
A.2	Sample DTE Modules	178
A.2.1	Base Module	178
A.2.2	Password Module	185
A.2.3	Ftp Module	186
A.2.4	Syslog Module	188
A.3	Excerpts of BLP PCP	189

A.3.1	Finding Paths	189
A.3.2	Relation Calculation	190
A.3.3	Pre-Apply	190
A.3.4	Post-Apply	191
A.4	Excerpts of MBLP PCP	191
A.4.1	Relation Calculation	191
A.4.2	Pre-Apply	192
A.4.3	Post-Apply	193
B	LMBench Results	194
B.1	Stock Kernel	194
B.2	LSM Kernel Using Dummy Module	195
B.3	LSM Kernel Using Capabilities Module	196
B.4	LSM Kernel Using DTE Module	198
B.5	LSM Kernel Using DTE and Capabilities Module	199
	Bibliography	201

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Phil Kearns, for his guidance, and my committee for valuable input.

List of Figures

2.1	Data Leakage with BLP <i>ss-property</i> alone.	8
2.2	Security rings	9
2.3	Sample ACM for 7 users and 30 files.	13
2.4	Sample ACL corresponding to above ACM.	14
2.5	Capabilities representing same access rights above.	15
2.6	Sample Unix file system.	19
2.7	Linux VFS-related kernel structures.	26
2.8	An example of a stackable file system.	27
3.1	Inodes and corresponding mapnodes.	47
3.2	A DTE policy to protect from <i>wu-ftpd</i> , with line numbers added.	67
3.3	Error messages resulting from attempted <i>wu-ftpd</i> exploit.	67
4.1	LMBench results for file deletion	71
4.2	File execution times for varying numbers of gateways.	78
6.1	Adding new domain specification rule in DTEedit.	94
6.2	Viewing a domain specification in DTEedit.	95

6.3	An error pattern popup warning in DTEview.	96
6.4	The file-manager tool in DTEview.	97
6.5	Domain transition analysis in DTEview	98
6.6	A reachability query in DTEview.	100
7.1	Algorithm to calculate BLP < relation	108
7.2	BLP for policy excerpt with disjoint type group.	111
7.3	Sample Assured Pipeline	114
7.4	Policy excerpt defining Clark-Wilson policy.	115
7.5	Policy excerpt implementing an assured pipeline.	118
7.6	Algorithm to calculate the modified BLP relation	119
7.7	BLP less than relation graph for base policy.	120
7.8	BLP less than relation for password policy.	120
7.9	Modified BLP less than relation for the password policy.	121
8.1	Sample dpc control file.	128
8.2	Priorities of access rules	133
8.3	A group declaration combining some untrusted domains	136
8.4	A possible definition of type root_t.	136
8.5	A sample dpc patch file.	137
9.1	Modification to DTE setup to store root namespace and prevent its unloading.	153
9.2	New DTE function to descend pathname using root namespace.	154
9.3	Modification to DTE hierarchical mount information setup.	155
9.4	Protocol to provide server authentication	160

9.5	The code to export DTE types from NFS server.	165
9.6	Code to import DTE types into NFS client from the network.	166
9.7	NFS client code to copy DTE types into inodes at first read.	167
9.8	NFS client code to insert DTE types into inodes on refresh.	168

ABSTRACT

Domain and Type Enforcement (DTE) is a simple and well-known access control system, which has been used at the microkernel level in SPIN, the kernel level in Unix, and the user-space library level in CORBA. This work implements DTE as a Linux Security Module, and provides tools for the composition and analysis of policies. The goal is to bring Mandatory Access Control in Linux to the level of ease of use of cryptography tools and libraries.

Tools have been created to edit DTE policies and query transitions through different privilege levels. A subtle modification of the Bell LaPadula (BLP) access control model's star property, applied to a DTE policy, results in a relation on types which permits us to concisely express, and therefore verify, goals for that policy. Policy creation is simplified using composition of policy modules, and enhanced by automatic verification of persistence of any desirable properties, including the modified BLP relation on types, across module application.

Domain and Type Enforcement in Linux

Chapter 1

Introduction

Access control in Unix systems, though certainly better than that in many other popular operating systems, leaves much to be desired. In particular, the combination of a trusted user, lack of mandatory access control, and far too many services running under the trusted user's id, are partly responsible for the large number of security-related advisories for Linux and other Unix systems.

Domain and Type Enforcement introduces mandatory access control to Linux, assigning labels to subjects and objects, and enforcing an access policy for all subjects, including the trusted user. It thereby greatly increases the potential for security in Linux systems.

1.1 Contributions

Research into improved OS access control is certainly far from stagnant. However, most of this is just that - research. This work addresses a deficiency in real, usable, yet complete mandatory access control systems.

First, an existing and well understood mandatory access control system, domain and type enforcement (DTE) [2], is implemented as a run-time loadable module for Linux. Rather than requiring even a cursory understanding of kernel compilation and installation, this allows anyone to install and begin using DTE with very little preparation. Furthermore, since we paid careful attention not just to correctness, but also to efficiency concerns, the implementation is not only useful, but also does not negatively impact performance.

Next, a set of administration tools has been implemented. This allows a system administrator to control a rather complicated access control system without having to study the syntax of the policy files. A brief tutorial will be sufficient to explain DTE policies and their administration. This tutorial also has been created.

Finally, the administration tools have been engineered so as to aid in the validation or refutation of invariants.

This work therefore presents a complete mandatory access control system, providing the necessary tools for system or security administrators to create, analyze, validate and finally implement security policies.

1.2 Organization

Chapter 2 provides background on the history and state of the art of systems security. Chapter 3 discusses our implementation of DTE for Linux. Chapter 4 discusses the performance of our implementation. Chapter 5 presents a formal analysis of the access rights of domains as restricted by a DTE feature, entry points. Chapter 6 discusses issues with policy administration, and presents a pair of tools designed to address specific problems

with editing a textual policy file. Chapter 7 presents a method for formal analysis of DTE policies. Chapter 8 presents a tool for constructing policies from a set of small modules, and uses the method presented in Chapter 7 to provide automatic enforcement and maintenance of any security properties across module application.

Chapter 2

Background

2.1 Security Nomenclature

Throughout this work, we will use the following nomenclature. Subjects are entities which can perform actions. For instance, a user is a subject. Processes started by the user are also subjects. Objects are generally file system objects, however they can be anything to which a subject can receive some sort of access. Subjects can therefore also be objects, as subjects can have access to each other. Security policies assign labels, representing some kind of security information, to subjects and objects, and determine access rights based upon these labels.

Most popular operating systems implement discretionary access control (DAC). They allow access rights to objects to be fully specified by the owners of these objects. Typically, an object is owned by the subject who created it. This has some major shortcomings. For instance, it facilitates Trojan horse attacks, where code, pretending to be friendly to the user, quietly gives away the user's access rights. Since the code is run as the user, who has the power to give away access rights to objects he owns, this is perfectly legitimate.

Mandatory access control (MAC) enforces a system-specified security policy which users

cannot modify [41]. For instance, a MAC policy might prevent users from giving away write access to their own objects. Ideally, MAC and DAC should be combined such that users can dictate access policies to their own objects, but within reasonable and safe limits.

2.2 Access Control Policies

An access control policy defines labels, subjects, objects and permissions, and dictates how and when labels are assigned to subjects and objects, as well as how and when permissions are decided and enforced.

This section reviews some traditional mandatory access control policies. The earlier are designed for security — preventing unauthorized data access — while the later address integrity — preventing unauthorized users from corrupting data. This reflects the historical shift, caused by a shift from military to business interests motivating research.

2.2.1 Bell-La Padula (BLP)

Bell and La Padula[4], while using a formal model to study the Multics system, introduced an access control policy intended to enforce the military security policy. This policy requires that no subject may read data classified at a higher security level than its own.

Military systems define a security level as a pair (l, C) , where l is a security level, and C is a set of categories. A category can be any kind of label which is meaningful within the context of the policy. A partial order is imposed upon security levels as follows: A security level $L_1 = (l_1, C_1)$, is said to dominate another level $L_2 = (l_2, C_2)$ provided that $l_2 \leq l_1$, and $C_2 \subseteq C_1$. This is written as $L_1 \propto L_2$.

In order to properly enforce the military security policy, BLP maintains properties. The first property, known as the “simple security property”, or *ss-property*, states the obvious goal:

ss-property: If subject S is granted observe access to object O , then $L(S) \propto L(O)$.

In other words, if S may observe object O , then its security clearance must dominate, that is, be greater than or equal to, that of the object. By itself, this property permits two subjects S_1 and S_2 to violate the military security policy through collaboration. Assume there is an object O_1 , such that

$$L(S_1) \propto L(O_1) \propto L(S_2).$$

In this case, S_2 is not allowed to observe O_1 , but S_1 is. By itself, the *ss-property* permits S_1 to leak the information contained within O_1 to subject S_2 . To accomplish this, S_1 copies the data from O_1 into an object O_2 such that

$$L(S_1) \propto L(O_1) \propto L(S_2) \propto L(O_2).$$

This scenario is displayed graphically in Figure 2.1. To prevent this security policy violation, BLP also contains the **-property*¹:

**-property*: If subject S_1 is granted observe access to object O_1 , and modify access to O_2 , then $L(O_2) \propto L(O_1)$.

¹Pronounced “star-property”.

This property prevents the situation displayed in Figure 2.1 from occurring. Once S_1 has read the data contained in O_1 , any object which it subsequently creates or modifies will have a security level at least as great as that of O_1 . S_2 is thus not permitted to read the data copied to O_2 .

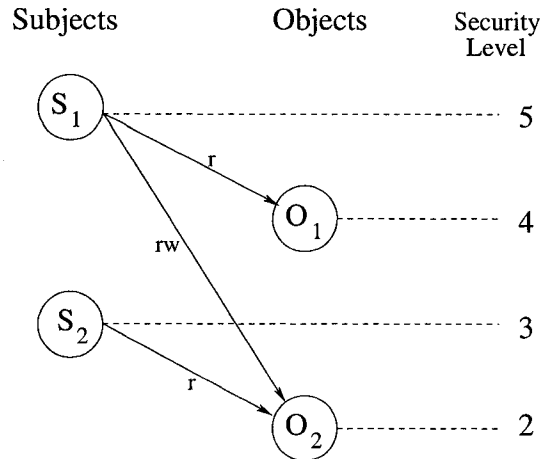


Figure 2.1: Data Leakage with BLP *ss-property* alone.

2.2.2 Rings

A ring-based policy specifies N concentric rings of protection. Privilege increases toward the center of the rings, with the center ring, known as ring 0, being the most privileged. Every object and subject is located within a particular ring. Subjects may not access objects within a deeper ring, that is, objects with a lower security level. A process changes its ring level by making a call to a procedure within a different level. However the process is associated with a lower bound, below which it may not pass. Furthermore, moving to a lower ring level is only allowed for certain entry points. The MULTICS operating system used ring-based access control[48].

The Intel 386 architecture is also a ring-based system, using four rings, referred to as privilege levels [28, Chapter 9.5]. Popular operating systems such as Linux, however, use only two [7, Page 37]. Ring 0 implements the kernel or supervisor mode, while ring 3 usually implements user mode. User mode instructions are not allowed to rewrite supervisor mode code. A system call is a call to code located in ring 0, so that, only for the duration of the system call, the privilege level drops to level 0. In this way, the kernel (or the operating system) is protected from user software. At the same time, it gives the kernel the power it needs allow multiple programs or processes in user-mode, while protecting them from each other.

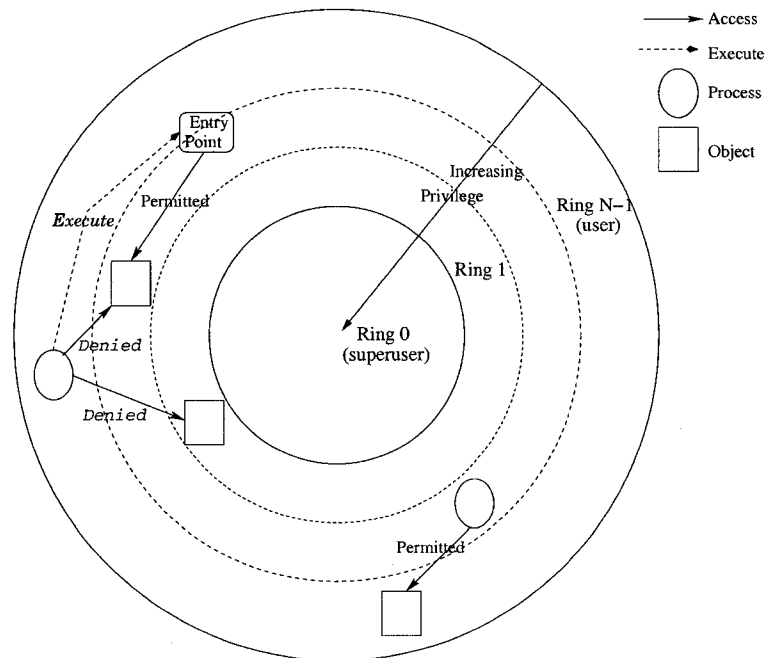


Figure 2.2: Security rings

2.2.3 Clark-Wilson

Most early work in access control was sponsored by the military. As such, the work centered around secrecy, which is the primary concern of the military. Clark and Wilson pointed out [9] that for the sake of commerce, integrity is at least as important as, perhaps more important than, secrecy. They created an access control policy to provide integrity, and compared its requirements to those of secrecy systems.

The policy which they presented was based upon three definitions.

- Data with whose integrity we are concerned will be called Constrained Data Items (CDI).
- Integrity Verification Procedures (IVP) are procedures which verify the initial state of a CDI.
- Transformation Procedures (TP). Given a valid initial state of a CDI, A TP transforms it into another valid state.

The system itself ensures that only certain users, under certain conditions, may execute a given TP, and that only some TPs may be used to modify a CDI. However, a large part of the policy exists outside the system, in the form of verification that the TPs and IVPs are correct, as well as the lists of users which may invoke TPs and lists of TPs which may alter CDIs. This means that with each software upgrade, any updated TPs or configuration files must be revalidated, a potentially costly proposal. In contrast, in a system which has been verified to satisfy the BLP policy, only an upgrade of the operating system requires revalidation.

To understand why, one must consider that, in BLP, the OS defines the entire access control policy. In a Clark-Wilson integrity control system, the TPs, CDIs, and lists of users permitted to use TPs and lists of TPs permitted to modify CDIs, are each a part in defining the policy. That is, these files are a part of the trusted computing base (TCB), and proving the correctness of any system requires verification of the TCB. We start to recognize a trade-off, then, between the power offered by a more malleable TCB, and the work involved in its verification.

2.2.4 Strict Integrity

Prior to Clark and Wilson, Biba attempted to address integrity using the inverse of BLP [5]. Once a subject S reads an object O_1 , it is no longer allowed to write any object O_x : $L(O_x) > L(O_1)$. In this way, users, or programs running on their behalf, are prevented from contaminating data with less trusted data.

This policy does not provide a method for taking in user data. Clearly, user data must be considered low integrity. Raising its integrity would have to be done by a trusted process. While this may sound reasonable, Clark and Wilson point out that a trusted process is one which is outside the integrity policy. Going outside the integrity policy to handle any type of user input is unacceptable. In the Clark-Wilson policy, the Trusted Procedures which handle this work are made an integral, and verified, part of the integrity policy.

2.2.5 Type Enforcement

From the work of Clark and Wilson, it may be observed that while security levels are best associated with subjects, integrity levels are better associated with programs (TPs).

Type Enforcement was introduced by Boebert and Kain of Honeywell [6] in 1985 as a method of implementing integrity systems without relying on a trusted user. It labeled objects as well as subjects, and specified access from subjects to objects, and from subjects to other subjects, in two matrices. Subject labels were called domains, and object labels were called types. Subject to subject access consisted of subjects transitioning to other domains. Domains were associated with procedures. For any procedure, a subject, which consisted of a procedure running in some domain, would be allowed to execute the procedure and remain in the current domain, execute the procedure and enter another domain, or not be allowed to execute it at all.

Subject to object access could be read, write, and execute. Type Enforcement was implemented first in the Secure Ada project (LOCK), and later by TIS in Trusted XENIX [1]. Secure Computing still uses TE in its Sidewinder firewall product [10].

Assured pipelines were introduced as an aside during the introduction of Type Enforcement. An assured pipeline is a non-bypassable subsystem through which data must flow between a particular source and destination. Boebert and Kain listed three requirements for demonstrating the security of an assured pipeline.

1. The subsystem which the pipeline attempts to enforce is indeed non-bypassable.
2. The transformation applied by the subsystem cannot be reversed or modified after the pipeline.
3. The subsystem is correct.

Assured pipelines are useful for proving that the transfer of data between security or integrity levels (or labels) is controlled.

2.3 Policy Representation

In Section 2.2, we presented several access control policies. Here we present the common methods of representing access control policies in the literature and in operating systems.

2.3.1 Access Control Matrix

In an access control matrix (ACM, see Figure 2.3), each row represents a user, and each column represents an object. An entry located at row u and column o specifies the access which u is granted to o . While conceptually simple, ACM's are not used in implementation because the matrices become very large and sparse, wasting valuable memory. However, they are frequently used to explain, and to make formal arguments about, access control policies.

		Objects (files)			
		f1	f2	...	f30
Subjects (users)	u1	r	r		rw
	u2	rw	rwX		x
	u7	r	λ		λ

Figure 2.3: Sample ACM for 7 users and 30 files.

2.3.2 Access Control List

An access control list (ACL, see Figure 2.4) is an abbreviated version of an ACM. At each object is stored a list of all users who may access the object and the types of access permitted

for each. Therefore, if only one user may access an object O , then the list need only contain one subject, not all subjects defined for the system.

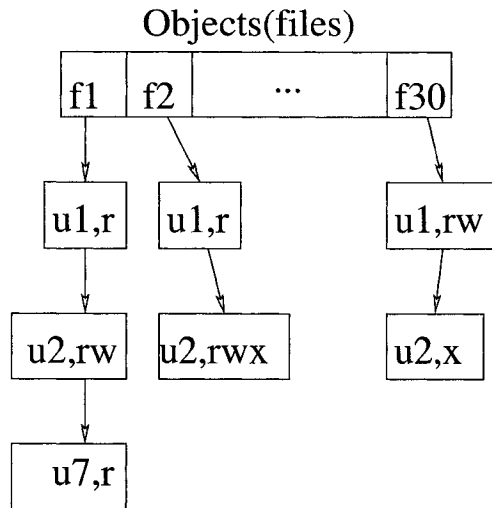


Figure 2.4: Sample ACL corresponding to above ACM.

2.3.3 Capabilities

The use of an ACL can be viewed as splitting an ACM into its columns, and then compressing these. Capabilities are sometimes described as doing the same thing by rows.

A capability [14] is a pair $\{o, r\}$ where r specifies a set of access rights to object o . A process has a list of these capabilities, and the union of the pairs in the list specifies the full access rights of a process. See Figure 2.5 for the running example expressed in terms of capabilities. A process may create, destroy, modify, and grant capabilities to other processes. While the use of capabilities makes for a very flexible and powerful system, it has some problems. One is the difficulty of discovering which processes possess a certain capability, and, by extension, of tracking the propagation of capabilities. Most real operating systems

therefore do not use capabilities.

Capabilities are, however, used in modern extensible systems such as SPIN [39]. In SPIN, capabilities are used to provide both extensibility, and security between extensions. Capabilities in this case are implemented through namespaces. Extensions call each other's functions by raising events, and provide functionality by registering event handlers. For instance, a memory management extension might save information to disk by raising an event which causes a file system extension to save the information. By binding a new event handler to this same event name, another extension can extend, or limit, functionality. For instance, a compression or encryption extension could extend the file system extension's write event handler by processing the data before passing it along to the file system extension's event handler. An extension cannot receive functionality which cannot be requested by raising some event. In other words, it cannot cause actions without the appropriate capability.

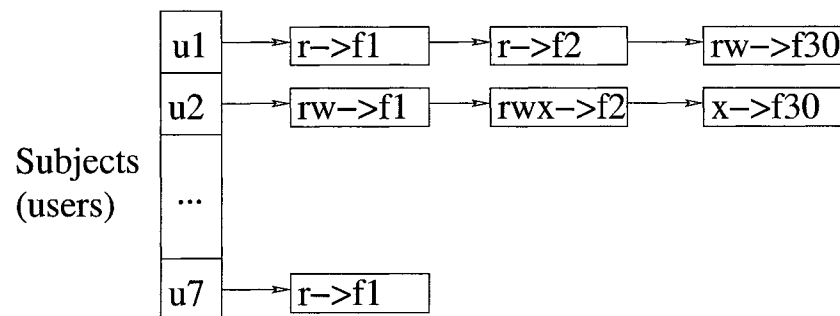


Figure 2.5: Capabilities representing same access rights above.

2.4 Role Based Access Control

In discretionary access control (DAC) systems, policies are specified in terms of rights to objects granted to users and groups. The user who creates an object usually owns it, and hence possesses all rights to that object. The owner may grant access rights to other users.

This is not how things are usually done in real life. Rather, subjects assume certain roles. For instance, the originator of a document may not be the author, or the owner, or the one who should be able to grant access rights to that document. All of these are normally assumed to be true in computing systems. Role based access control (RBAC) attempts to make computing systems resemble real life access systems more closely [47].

Roles are used in two ways. In SELinux [33] and many other RBAC implementations, they are simply used as a “hat” which a user wears in order to be granted extra privileges. When most people speak of role based access control, they think of this very simple interpretation.

Clark and Wilson [9] presented an access control system where roles are used to implement separation of duties as required in common accounting practices. As described in Section 2.2.3, CDIs may only be modified by a set of TPs. The TPs must be executed by some combination of roles. For instance, a cashier and a manager might both be required to be present in order to modify the amount in the register. This means that any two (different) people who may assume these roles may come together to run the TP, in order to modify the CDI, but neither may do so alone. This is a powerful concept. However, it is sufficiently complicated that it is implemented only in proprietary software, aimed mainly at banking institutions. Beyond the complexity of creating the operating system itself, one

must also deal with maintaining such a system. In order for this system to be secure, the membership of all roles must be verified, as must the list of roles required to authorize a TP. Furthermore, each TP must be verified *each* time the code might change.

2.5 UNIX

In Unix systems, access rights to objects are specified for users and groups, and stored with the object. Usually, every person who will use the system is assigned a unique user id. Each user is associated with at least one group. The default group is listed in the user definition, which is found in the password file (`/etc/passwd`). A user can be placed in additional groups by placing the username in the group definition in the groups file (`/etc/group`). A running process carries along its real and effective user and group IDs.

2.5.1 File System

The UNIX file system is based upon file descriptors, called inodes. Inodes can be uniquely reference by an integer, known as the inode number, and the file system upon which they are located. All file objects, including directories, devices and normal files, are represented as inodes. A directory is a file associating file names with inodes. For instance, a directory might associate the following:

File Name	Inode Number
file1	32079
file2	32167
file3	32254
dir1	33152
file4	32056
file5	32254

Notice that `file3` and `file5` are associated with the same inode. This file can be referenced using either name. It may also be associated with other names in other directories, and will not be deleted until all names are removed. However, since an inode number is unique only to the file system on which it is located, all names which are associated with this inode must be located on the same file system. Every directory contains at least two entries, `“.”` and `“..”`. These always refer to the directory itself, and the parent directory, respectively.

A Unix system starts with a particular disk partition mounted as the root file system (`/`). Other disk partitions can then be mounted on top of any existing directory. Mounting a partition on a directory places the root of the file system located on the new partition at the specified mount point. For instance, if a partition `/dev/hda3` is mounted on `/usr/local`, then any request to access a file under `/usr/local` will look up the part of the path-name after `/usr/local` on the mounted partition. If any files or directories existed under `/usr/local` on the root partition, they are now hidden until `/dev/hda3` is unmounted. This allows all file systems to be viewed as one large tree.

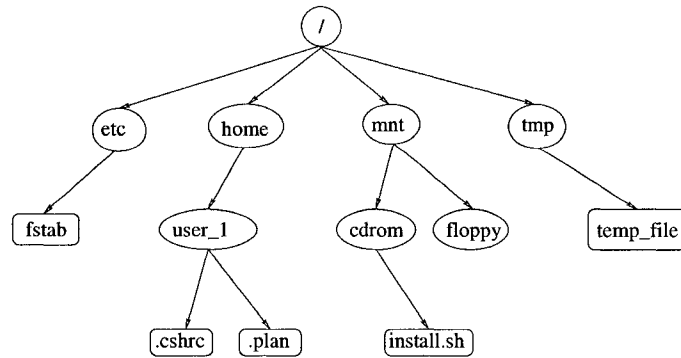


Figure 2.6: Sample Unix file system.

A sample Unix file system tree is shown in Figure 2.6. In this figure, for example, the file `/mnt/cdrom/install.sh` is on the cdrom file system, yet this fact is entirely unimportant to users on the system, as the file appears just like any other.

2.5.2 File Access

Unix permissions do a remarkable job of allowing for great expressiveness using a minimal amount of space. As mentioned above, a Unix system recognizes individual user ids and specified groups of users. People who wish to use the system are assigned a user id and default group. Each user may belong to several other groups as well. Each file is assigned one owning user and group. Access permissions are then specified using 12 bits. Basic permissions consist of read, write, and execute permissions, specified individually for the user owning the file, the group owning the file, and the rest of the world. Since a file can be a file or a directory, these permissions are multiplexed as follows. Read permission on a file permits viewing of the file contents. Read permission on a directory does likewise, but the contents of a directory are the names of files under that directory. Similarly,

write permission on a file allows modifying the file's contents, whereas write permission to a directory is required to rename, create or delete a file under that directory. Execute permission on a file is self-explanatory, but execute permission on a directory is interpreted as the right to descend that directory — that is, to view the contents of files the directory contains, subject to the individual file permissions. Therefore it is possible to allow reading a file's contents, but not its name, and vice versa.

Three more bits are used for file permissions. One is the sticky bit, which has two interpretations. First, a program whose sticky bit is set remains in swap after termination. Second, a file under a directory whose sticky bit is set may only be deleted or renamed by the owner of the file or the owner of the parent directory, regardless of access permissions. The other two bits are `setuid` and `setgid`. Executing a `setuid` program file changes the process' effective user id to that of the file owner. The `setgid` bit does the same for the process' group.

2.5.3 Signal Access

Unix processes can communicate by sending each other signals. Some signals force a process to be suspended or killed, while others can be ignored or handled by the process' own signal handlers. A process is allowed to send signals to other processes owned by the same real or effective user id. Under Linux, a signal may also be sent to any process under the same process session, that is, between processes sharing the same controlling terminal.

2.5.4 Superuser

There is, in most systems, the notion of a superuser, named *root* in Unix systems. The superuser may access any file, send any signals, and change any permissions. In order to limit damage due to the compromise of a system, superuser file access can be limited for remote file systems, but, for ease of remote administration, it often is not. Obviously, the compromising of superuser on a system renders the system entirely untrustworthy, as the attacker can do anything he likes, including replacing system monitoring tools with versions hiding his own activity. Patching the vulnerability exploited by the attacker is insufficient since anything else may have been damaged, so the only solution is to rebuild the system from original media and backups predating the attack. Unfortunately, more and more services are offered by most machines, and these usually require access to privileged files or services. If any of these services are compromised, the attacker becomes the superuser and hence owns the system.

2.5.5 POSIX Capabilities

The nature of the superuser, that one either has all its powers or none, is a major problem. POSIX capabilities [25] attempt to solve this by splitting the superuser's powers into a set of distinct capabilities, such as the ability to open a restricted port (< 1024), modify the network configuration, or trace any process. A process is created with a set of permitted capabilities, and may further restrict these at will before starting a new process. For example, the *talkd* service may only need access to restricted network ports, so that it may be started with only the `CAP_NET_BIND_SERVICE` capability. If *talkd* is later compromised, the attacker's privileges on the system are still very limited, despite being root on the

system.

POSIX capabilities are similar to Dennis and Van Horn's classical capabilities [14] in the way they relate to a process and can be granted or given up individually. They differ in that POSIX capabilities do not designate rights to objects, but rather specify generic subsets of the superuser's powers.

Linux partially supports POSIX capabilities [30]. Processes carry three bitmaps, representing the Inheritable, Permitted, and Effective capabilities. Executable files will also carry capabilities, but the Linux VFS does not yet support them. In the meantime, a file's capability sets are assumed to be empty, barring two exceptions. If the user executing the file is root, then the file's Inheritable set is full, and the file's Permitted set is full except for CAP_PSET, which permits granting capabilities to other processes. The same is done if the file is setuid root, and, in addition, the file's effective set is full.

Equations 2.1 through 2.3 show how capabilities are computed upon file execution. Here pX denotes a process attribute X , and fX denotes a file attribute X . While pX denotes the attribute before the file execution, pX' denotes the same attribute as it was recomputed during file execution, and as it will be applied for the remainder of this execution. The attributes are P for the permitted set, E for the effective set, and I for the inheritable set of capabilities.

$$pI' = pI \tag{2.1}$$

$$pP' = fP \vee (fI \wedge pI) \tag{2.2}$$

$$pE' = pP' \wedge fE \tag{2.3}$$

Clearly capabilities cannot be computed correctly until file capabilities are implemented. In the meantime, however, the code supporting capabilities is implemented in the rest of the kernel, and this code is now used for implementing other features. For instance, calling `setuid(0)`

currently sets all capabilities for the calling process, and a process which attempts to use `nice` to adjust the scheduling priority for another process is checked for the `CAP_SYS_NICE` capability, rather than for the effective uid of 0 [7, Pages 556-558].

2.5.6 Domain and Type Enforcement

Domain and Type Enforcement was first presented by O'Brien and Rogers [37], and was an extension of Type Enforcement, presented in Section 2.2.5. It differed from TE in part by specifying policies in an intuitive policy language rather than using two matrices. TIS based the first Unix implementation of DTE [2] on OSF/1 MK4.0. Their domain transition semantics were somewhat different from those in TE. A domain transition in TE occurs every time a domain executes a file for which the security policy mandates a transition. In DTE, a second, voluntary type of transition is added. The mandatory transition is called an auto transition, while the voluntary transition is called `exec`. If a domain D_1 has auto access to D_2 , and a process under D_1 executes a file which is an entry point to D_2 , then the process is automatically switched into domain D_2 . If domain D_1 has `exec` access to D_2 , and a process under D_1 executes an entry point to D_2 , the process by default remains under domain D_1 . However, if it so requests, it may, on the same execution, be switched to D_2 . Voluntary transitions are useful for programs, such as `login` or `sshd`, which may need to switch to one of several domains, depending upon the credentials presented.

Mandatory transitions are useful both for restricting untrusted programs, and for fitting legacy programs into a DTE policy without requiring any rewriting or recompilation.

DTE is designed to fit elegantly with the concepts of UNIX. Object labels are assigned hierarchically, in a structure mirroring the file system tree. Types are assigned to pathnames using either explicit or recursive rules. An explicit type assignment rule assigns the type only to the pathname, whereas a recursive type assignment rule assigns the type to all the pathname's descendants.

At the same time, a clean separation is maintained between the UNIX DAC and DTE's MAC. The policy type assignment rules interpose a layer between DTE types and UNIX files. Domains transitions are performed only at file execution, and have nothing to do with UNIX users. Creating relationships between users and domains is thus left to user-space programs, such as `login` and `su`, or PAM modules acting on their behalf. While some see this as a deficiency [38], we believe that leaving the amount and method of cooperation between MAC and DAC to the system configuration is one of DTE's strengths.

DTE was developed with the purpose of mediating access between users, files, and network traffic. The underlying concepts of DTE have also been applied at other levels. SPIN used DTE to protect kernel extensions from each other [39], while OO-DTE [35] applies DTE to a distributed object framework, CORBA.

2.6 Linux File System Architecture

Linux abstracts away file system specific details behind the Virtual File-System Switch (VFS). Applications call VFS functions, which in turn know how to deal with the real file

system.

Every file and directory in use is represented by a VFS inode, which holds the relevant metadata, such as file size and access permission, as well as pointers to file system specific functions to operate on the file or directory. In this way, the VFS need know nothing about how to actually open a file on the file system in use. In fact, a file system can be provided as a module, simply telling the Linux kernel what functions to call for applicable operations.

An inode is the operating system's representation of file metadata. A process, however, needs to have its own representation of files, able to store some data which may be different from another process' for the same file. The file structure includes a pointer to the inode, as well as data regarding access permissions with which the file was opened, and a pointer into the file representing the current position.

The inode has a pointer into the directory cache. The directory cache is a hash table of structures called dentries, each of which contains a path component's name, as well as pointers needed to construct pathnames, such as the parent directory and, if this is the root of a mounted file system, the covered directory.

A `vfsmount` structure contains data regarding a mounted file system or VFS sub-tree, including pointers to the mount point and root of the mounted file system, and other mounts of the same file system. It is the glue which holds together one tree constructed from many file systems, as well as folds within itself (as will be seen later). A superblock contains information about a block device containing a mounted file system, and pointers to the `vfsmounts` which mount this device.

Figure 2.7 shows the use of some of these structures when the cdrom on `/dev/hdc` is mounted under `/mnt/cdrom`, and contains files `README` and `FILES` in its root directory. The

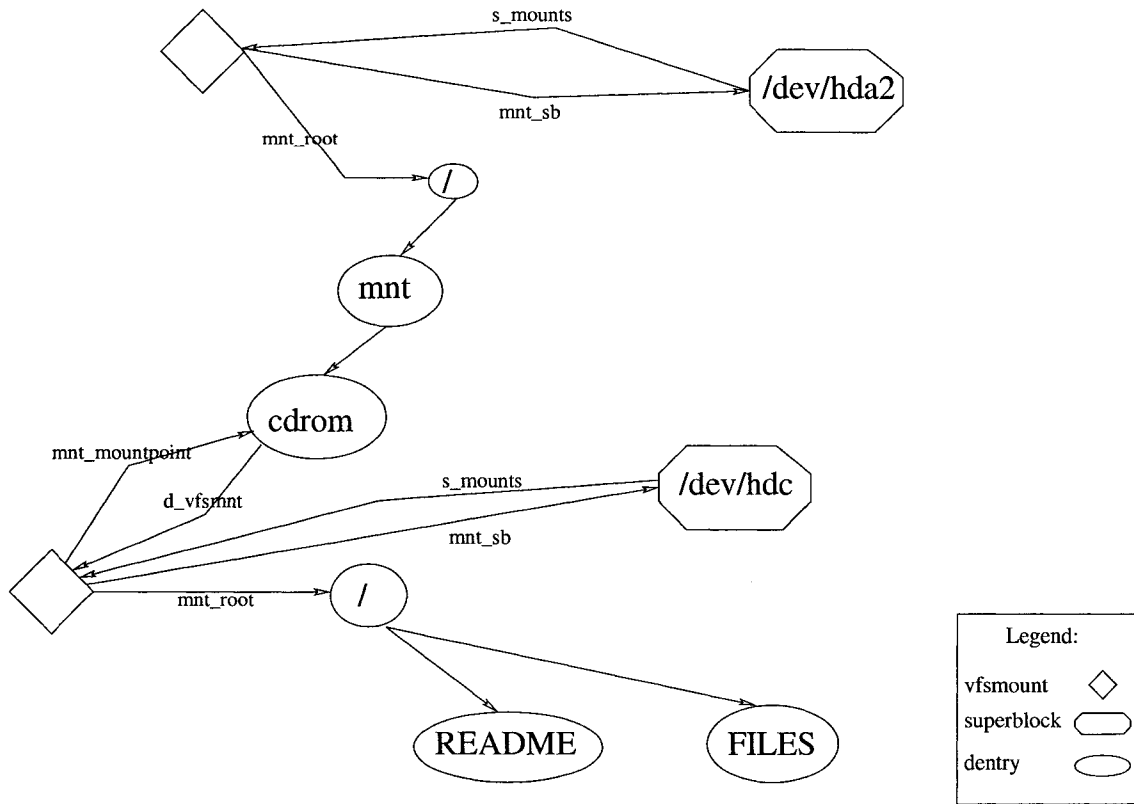


Figure 2.7: Linux VFS-related kernel structures.

links represent pointers between the various structs. For instance, the dentry representing /mnt/cdrom has a d_vfsmnt pointer set, indicating that a file system is mounted on top of the dentry. The value of the pointer is the address of the vsmount struct representing the mount instance of device /dev/hdc. The vsmount struct in turn points back to the dentry on which it is mounted.

2.7 Stackable File System

As noted above, every VFS inode in Linux contains pointers to the file system specific operations to be used on the associated file. Stackable file systems increase the levels of

indirection, allowing programmers to introduce a feature to any file system in the form of filters rather than by having to modify the file system itself. Zadok has introduced stackable file systems to Linux as well as FreeBSD and Solaris [17]. The most obviously useful application of stackable file systems is the transparent implementation of encrypted file systems. Figure 2.8 demonstrates the use of stackable file systems. On receiving a write request, the kernel calls the stackable layer first, which, in this case, encodes the file data in some way. It could also encode the filename or file attributes. After this, the write function for the file system upon which the file is located is called. A read is performed in the reverse order, calling the file system specific read function first, then calling the filter specified by the stackable file system to decode the data read from disk.

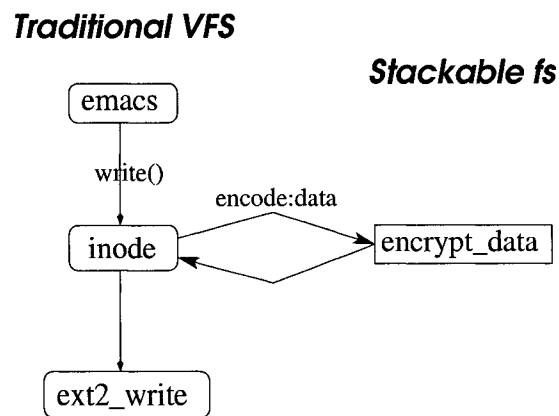


Figure 2.8: An example of a stackable file system.

A stackable file system can be mounted in one of two ways. It can be mounted as an *overlay*, such that the mountpoint of the decoded file system is on top of the encoded file system. In this case, after the mount command, the encoded file system is no longer visible. The stackable file system can also be mounted at a different mountpoint. In this

case files can be accessed either decoded, through the new mountpoint, or encoded, through the original mountpoint.

Increasing the power of stackable file systems even further, FiST, a new language for defining stackable file systems, allows things such as fan-in and fan-out, where one block seen by the VFS could correspond to many blocks on the underlying file system, or vice versa [57]. This could be used, for instance, to implement a RAID file system layer, or a file system compression layer.

2.8 Networking

It is taken for granted today that computers should be networked to allow communication (between people), data sharing, and often sharing of processor and memory power. For far too long, however, network security has been mostly dependent upon the goodwill of users. For instance, it has been only recently that ssh, or secure shell, has begun to replace telnet for connecting to remote machines, even though telnet sends passwords in plain text, so that any user able to listen to network packets meant for other processes and other machines can read these passwords.

Type Enforcement and DTE have been applied to network packets as well as files [10, 53]. Every network packet is labeled with a type. This type can be determined by the domain from which it was generated, or, for firewalls, by the network (card) from which it comes. TIS used this to implement DTE-enabled NFS. The NFS server, then, can determine the type of files it exports, rather than rely on the client's policy to be synchronized with its own.

2.9 Other Projects

Much recent work in systems security implements very simple security policies. An example of this is LOMAC [22], which is basically a two ring system (see Section 2.2.2) intended to separate services under Linux from user processes. Another example is the HP Trusted Linux system [13], which compartmentalizes services such that they cannot interact. In this way, compromised services have no ability to compromise one another.

Other systems security projects offer versions of existing OS distributions which have been modified to address particular known weaknesses. Examples of these are the Immunix System [18], a version of RedHat Linux offering protection against stack smashing attacks and format bug exploits, as well as an enhanced ability to restrict system access by services, and OpenWall, which does not offer its own distribution, but a set of patches to protect against very specific vulnerabilities [15].

In the following sections, we discuss a few recent and current projects in more detail.

2.9.1 Linux-ACL

Unix file permissions can only specify read, write and execute permissions for the owning user, the owning group, and the rest of the world (see section 2.5.2). The Linux-ACL project [26] does not elaborate on the types of access, but allows access specification for additional users and groups. This allows a user, for instance, to provide write access to another user without having to request a new group containing the two users, or having to give write access to a larger group or the rest of the world. Linux-ACL is now being implemented on top of extended attributes, which permit specification of arbitrary types of access by name.

ACLs (and extended attributes) present several problems, because they are not a part of the original file system. Neither backup utilities nor tools such as `tar` are able to preserve ACLs. However, the ACL information can be extracted into separate files and then reapplied.

Linux-ACL remains useful on a DTE system. Domain and Type Enforcement is designed to be a mandatory access control system and, as such, does not allow users to control the domain to type access allowed. ACLs allow a very convenient method for quickly and precisely specifying the needed access control to a user's files.

2.9.2 LIDS

The Linux Intrusion Detection System [56] aims to implement mandatory access control and improve intrusion detection through increased and improved kernel logging. Implementation of MAC may appear a separate problem from IDS, however without MAC any log created by the IDS is vulnerable to attack, rendering the IDS itself useless. Since intrusion detection is usually accomplished through logging, and minimal access reduces file vulnerability, LIDS adds *append* as a possible type of file access.

The contributions of LIDS sit between those of capabilities (see section 2.3.3) and DTE. Access rights may be specified by subject and object. The object can be a file or directory. The subject may be omitted, in which case the access rights apply for the whole system, or the subject may be an executable. These access rights may descend down the file system tree. Processes, however, do not inherit access rights from their parents, and access rights may not be passed to other processes as capabilities can, which limits the power of LIDS. For instance, since access rights are associated with an executable, different users starting

`/bin/bash` will receive the same access rights, whereas DTE or classical capabilities allow different situations to associate different access rights with instances of the same executable. However, as a quick tool to protect sensitive files or implement more robust logging, LIDS is very useful.

2.9.3 TE and DTE

Other teams are working on implementations of Type Enforcement and DTE. Secure computing still sells its Sidewinder firewall project which uses Type Enforcement [10]. TIS, now a part of Network Associates (NAI), still maintains its BSD-based DTE system [2], and is rumored to be starting a FreeBSD implementation. Finally, a project at Science Applications International Corporation (SAIC) has been working to implement DTE in Linux, although it appears either progress is slow, or updates are not being announced.

2.9.4 SELinux

SELinux is a joint project of NAI, the Secure Computing Corporation (SCC), and the National Security Agency (NSA), to implement the Flask [46] security architecture in Linux. The goal of Flask is to support a wide variety of security policies. It separates access control into two separate services. The Security Server stores access control policies and makes access decisions. The Object Manager enforces the policies. On an object create request, for instance, a client asks the Object Manager to create an object, who in turn forwards the request to the Security Server. Policies implemented under SELinux include Type Enforcement, Role Based Access control (RBAC), and Multi Level Security (MLS).

This project is important for many reasons. First, it shows that the government is

serious about system security. Second (resulting from the first), it should cause kernel developers to take improved access control more seriously. Third, since this appears to be a high level access control system — that is, it facilitates the implementation of access control policies, rather than dictating one — it will be a useful stepping stone for the NSA as well as others to continue research into security and access control.

However, its practicality remains to be seen. Whereas LIDS errs in being too simple and too limited in its ability to express security policies, SELinux may go too far the other way. System administrators should not need to spend weeks learning how to use a system to protect against a new *talkd* exploit. Hopefully, DTE will fit nicely between these two extremes.

2.9.5 HP-LX

HP has developed a distribution of Linux designed to minimize damage due to compromise of system services. They argue explicitly that, regardless of due diligence, systems will be compromised. HP-LX [16] provides features in order to contain both file access and communications by processes. This appears to be based upon the compartmentalization offered by their previous Trusted Linux [13] implementation. It also includes TripWire [21] to ensure the integrity of important system files; a script which performs several functions to ensure a safe system, such as remove *suid*/*sgid* bits from executables; and a secure audit daemon, which is protected within a compartment.

2.10 Other Work in Security Policies

Very little work has been published regarding security policies. Most systems security work has been aimed toward implementation of security enhanced systems, with little or no thought given to policy creation, analysis, and maintenance. A recent SELinux paper [44] explains the syntax of various textual policy configuration files in great detail, but does not offer any assistance beyond the textual policy specification. Another SELinux paper [34] presents a specific policy, and discusses its meeting several specific criteria, such as confinement of the sendmail service. It does not offer any formal specification of these criteria or the policy.

Fraser and Badger performed automated analysis of DTE policies [23] in order to allow safe updates to a running DTE system's policy. This is the best example of formal policy analysis to date, and we base some of our work upon this paper.

It is clear from the above review of recent and current work in systems security that there is a shortage of research into the subject of security policy configuration. This work aims to address that shortage.

Chapter 3

DTE

This chapter discusses our DTE kernel module implementation in detail. Since a goal of our project was to provide a mandatory access control system which is both easy to install and to administer, it naturally must be implemented for current kernels. Since Linux is a very quickly evolving operating system, this requires constant vigilance. Operations upon which an implementation is based at one moment can become wholly meaningless within a few kernel versions.

A prototype of our DTE implementation was presented at the Atlanta Linux Showcase (ALS)[27]. Many of the details in the implementation still hold true. However much has changed. This chapter presents the current implementation of DTE. Section 3.1 discusses the new framework upon which the DTE implementation is based. Section 3.2 discusses the fundamental decisions concerning the implementation. Section 3.3 introduces some important data structures, while Section 3.4 presents algorithms for fundamental DTE operations. Next we show the configuration file syntax and the DTE API in Sections 3.5 and 3.6, respectively. The chapter closes with a demonstration of the effectiveness of DTE in stopping real-world attacks against otherwise vulnerable systems.

3.1 LSM

The Linux Security Module (LSM) project is a response to Linus Torvalds' call [51] for a general framework to support security extensions. Responding to a paper [33] presenting the SELinux access control system, Torvalds announced that he would not accept any such patches into the kernel. Rather, he asked the security community to provide a set of truly generic security patches. These patches should allow all, or at least many, of the existing security projects to work as Linux modules, without requiring any further kernel patching. This would absolve Torvalds of having to make any decisions on a single appropriate access control policy.

Crispin Cowan took the initiative [11] in creating the LSM project [50, 54], which was joined by other noteworthy people such as Stephen Smalley and David Wagner. Many groups working on projects to extend access control in Linux were represented, including SELinux [33], Immunix [19], and projects by HP and IBM. Clearly, each of these groups had developed its own methods of dealing with certain problems. Every issue to be addressed by LSM, therefore, raised much discussion and required a great deal of compromise.

The LSM architecture, as recommended by Torvalds, became a structure containing pointers to functions which perform various security checks, as well as data initialization to support these checks. By default, the functions pointed to by this structure are dummy functions which default to permissive behavior. Wherever the kernel performs a sensitive operation, for instance opening a file with write permissions, a call to an LSM function is first made. If the LSM function returns an error, then the modification is not performed. A security module, when loaded, may redirect any or all of the LSM function pointers to

its own functions, in order to accomplish its goals.

Clearly these functions, and the places whence they are called, will affect what sorts of policies can be used. A significant part of the design of DTE for Linux 2.5 consisted of taking part in the LSM project in order to ensure that it was able to support DTE as a module.

3.1.1 LSM Design

As described above, the overall design of LSM is modeled after the design of other Linux subsystems such as the VFS [29]. The active security policy is represented by a set of function pointers. These function pointers are dereferenced and called throughout the kernel to make policy decisions. Inserting a new security policy therefore consists merely of redirecting these function pointers. A few other design decisions bear discussion, however.

First, since security policies are now loaded as modules, the question may be raised of what sort of policy is needed, or wanted, in the base kernel. On the one hand, we want the LSM project to minimize intrusion into the existing kernel code. This will maximize the chances of LSM being accepted into the mainstream kernel, which is, of course, the main purpose of this project. On the other hand, by simplifying the base Linux kernel's security policy, performance will improve. This will be a particular advantage for embedded Linux systems. In this case, the allure of this performance increase outweighed the increased code intrusion. POSIX capabilities [30], which have been a part of the Linux kernel since version 2.2, were removed from the kernel and are now provided as an LSM module. The Linux base security policy under LSM is therefore the simple "root is everything" which most sysadmins prefer.

An issue which has undergone much discussion was that of module stacking. The naive first impulse is to ask for the ability to load combinations of arbitrary modules. For instance, it would be advantageous to load the capabilities module along with the DTE module. However, arbitrary composition of security policies is not possible [24]. LSM's solution is to allow policies to define the `register` and `unregister` functions. Some LSM module will be the first to be loaded. Subsequent module insertion requests will be handled by this module. Therefore, if the author of the first module is aware of another module, and finds the other module to compose nicely with his own, then his module may permit the second module to load. If the second module implements an `inode_permission` function, for instance, then the first module's `inode_permission` function may first call the second module's function. Provided this function permits the requested action, the first module may perform its own check.

Another issue which the LSM project faced was communication of modules with user-space programs. Most existing enhanced access control policies, since they required patching the kernel anyway, simply introduced system calls. However, LSM could not simply reserve a large block of new system calls so that all LSM modules could have their own. In addition, the currently preferred method of providing communication between the kernel and user levels is to implement a small pseudo file system [31].

The solution implemented by LSM was to reserve a single system call, called simply `security`. This system call takes three arguments. The first is an integer named `id`, which identifies the LSM module with which the user-level program wishes to interact. The second is another integer variable `call`, which identifies the particular system call of the LSM module which the user-level program wishes to invoke. The third argument is a `void`

pointer, which may contain any data the module wishes to accept, plus buffers for return data. In general this will be a structure, which will itself point to several arguments. For an example of how this is used, see the details on the DTE API in Section 3.6.

3.2 DTE Design Decisions

3.2.1 Entry Types

DTE facilitates restricting file system and signaling access to precisely that needed for processes to accomplish their tasks. Further protection is afforded through entry points. Entry points are binaries whose execution may trigger or allow transitioning to another domain. A domain may be entered only while beginning execution of one of its entry points.

Our prototype DTE implementation, as well as the DTE implementation by TIS, specified entry points as lists of binaries. Our LSM DTE module, however, uses lists of entry types instead. There are several reasons motivating this switch. First, we no longer need to reconstruct the executable's filename to confirm its being an entry point. This in itself provides two benefits. The consideration of how mounting activity such as multiple mounts and per-process namespaces affects the actual filename is left entirely up to the file-type resolution subsystem, rather than needing to be duplicated locally at the DTE `auto` and `exec` decision algorithms. This both reduces the amount of code, and therefore the risk of dangerous bugs, and increases performance. It also prevents time of check to time of use races [45], where an attacker, or valid activity, is able to change an object between the permissions check, and the use of the object.

Additionally, the use of entry types is beneficial to policy administration for two reasons. When creating a new domain under the prototype DTE system, for instance `login_d`, it is tempting, and permitted, simply to define `/bin/login` as the entry point without giving it any further thought. Most likely, its type will be the same as that of all other files under `/bin`. This is the location of many binaries which are frequently updated. Write access under `/bin` will therefore likely be quite liberal.

By requiring the definition of an entry type, we require the separate specification of access to the `login` binary, which encourages providing minimal access to entry points, if only on account of laziness — providing more liberal access than needed provides no benefits and takes more work than providing the minimal access. Doing so also reminds the policy administrator of the importance of the entry points which, in the case of `login_d`, will likely encourage him to move `/bin/login` to `/sbin`, since no one should have create access to `/sbin`, but such access under `/bin` is, again, likely to be granted more freely.

Finally, in chapter 8, we will introduce a relation on types which provides a quick glimpse of the secrecy and integrity properties of types. When specifying entry types instead of entry points, the presence or absence of integrity for entry types, which is crucial to domain security, becomes immediately obvious. This information is of course also available when entry points are used, but obtaining it requires more work. A policy administrator would have to look up the type of each entry point to a domain and determine its place in the type relation. This extra work interferes with the concept of interface zen [8], by placing needless obstacles in the way of obtaining information.

3.2.2 File Type Resolution

DTE assigns types to files hierarchically. A DTE policy contains a set of *type assignment statements*. Each statement is of the form

```
assign -e type1 path1
```

```
assign -u type2 path2
```

```
assign -r type3 path3 path4
```

and binds a typename to one or more pathnames. The option indicates the type of binding. If it is “-e”, for “explicit”, then the type is assigned only to the specified pathname itself. If “-u”, for “under”, then the pathname is assumed to be a directory, and the type is assigned to descendants of this directory, but not the directory itself. The option “-r”, as shorthand for “-eu”, is also permitted.

TIS’ implementation of DTE used the policy’s type assignment statements to initialize the type assignments. However, a running system did not consult the type assignment statements when assigning types to newly created files or files which were moved. When a file was moved, it retained its original type, in effect creating a new type assignment rule. When a file was created, it was assigned a default type based upon the creating domain [3].

Our implementation of DTE works somewhat differently. We consider the above system to be too complex. In many cases there may be no way to predict the type assigned to a particular file. To discover the type, we must either know the entire history of file system activity, or query the DTE system. In our prototype DTE implementation, the type assignment rules were always followed. Therefore, the type assigned to a file was always predictable, given only the pathname. This presented two implementation challenges. The

first was a performance issue on a directory move. Since each descendant of the directory may be assigned a new type as a result of the directory move, the types cached for any descendants had to be forgotten. For a large directory tree, this could be time consuming. However, since the values can simply be NULL'ed, and reloaded when needed, the performance impact was seldom noticeable.

The second implementation challenge was the result of hard links. As discussed in Section 2.5.1, Unix presents all files, devices, and directories as files. Files themselves are usually a set of data blocks, but always organized by a descriptor called the inode. Inodes are simply referenced by a number unique on the file system. A filename, then, points to an inode number. Many filenames can point to the same inode number. Each filename is called a "link," and a file is not truly deleted until the last remaining filename referencing the inode is deleted.

When a DTE system opens a file whose DTE type is not cached, it uses the provided pathname to determine the DTE type. Clearly, since more than one name can refer to a file, the actual type assigned to a file depends on the name first used to open the file. Subsequent `open` system calls for the same inode will not recalculate the typename so long as the type information remains cached, even though a different process, under a different domain, may use a different pathname to open the file. The solution which our prototype implemented was to use one file for each file system to specify inode numbers pointed to by more than one name, along with the pathname which should be used to determine the inode's type. On an `open` call, if the inode being read was listed in this file, the pathname listed in the file was appended to the file system's mountpoint, and this pathname was used to determine the file's type, rather than the pathname provided by the user.

In the LSM DTE implementation, directory moves still incur the overhead of uncaching the type information for all descendants. The file system specific hard link file, however, is no longer supported, as the problem with hard links is solved by support for extended attributes.

As of Linux kernel version 2.4, the run-time hierarchical assignment of types to files met with another challenge. A file system can now be mounted more than once on the same system. Assume a file system containing a file named `file1` is mounted under both `/mnt/fs1` and `/scratch1`. DTE handles this in several ways. By default, DTE stores the first location under which a file system is mounted. If the file system is mounted under a second location, DTE will continue to use the first mount location as the base of pathnames under that file system, even if the first mount instance is removed. Only when all mounts of the file system are removed, will this information be released. Alternatively, the DTE policy may specify a pretend mount location for a device. When the file system stored on this device is mounted, DTE will always pretend it was mounted under the pretend path, regardless of the actual mount location. Finally, the DTE policy may forbid mounting of a device under any location other than one which is specified. This location is called the restrict location.

3.2.3 Extended Attributes

The LSM DTE implementation supports file-type bindings through extended attributes. Each file system with persistent inodes may contain a file at its root named `dteeaf`. This file is initialized using `DTEedit`, the policy creation tool presented in Section 6.1. It specifies the type for every inode number up to the maximum allowed inode number on the file

system. Since this number is often very large, we conserve space by creating a table for each file system relating the type name to an index, which can be as small as 1 byte¹. By reserving space for nonexistent inodes, we eliminate the need to search for an inode entry in the file. Rather, we can calculate the position of an entry in constant time as

$$p = t + s \times i,$$

where t is the offset of the beginning of the inode table, s is the size of an index entry, either 1 or 2 bytes, and i is the inode number. Furthermore, when a new file is created, there is no need to shuffle existing entries to keep the table sorted.

Since extended attributes establish a correspondence between types and inode numbers, rather than between types and pathnames, the hard link problem is automatically solved. Extended attributes are purely optional. If the file `dteeaf` exists at the file system root, then the extended attributes table is automatically read and maintained. If not, then we rely on the traditional hierarchical method of type assignment.

While extended attributes are a useful enhancement to the DTE implementation, we wish to maintain the strictly pathname based type assignment. Therefore, when a new file is created on a file system supporting extended attributes, its type is determined in the same way as for a file system which does not support extended attributes. The calculated type is then stored in the `dteeaf` file.

¹This is dependent only upon whether there are fewer than 128 defined types.

3.2.4 Policy Updates

A user friendly system would allow unloading of the DTE module, rereading of the policy configuration file, or loading of policy updates. However, each of these options may be unsafe. If an attacker is able to update the security policy, the system cannot be truly secure. There may be situations where updates are acceptable. For instance, if it can be proven that updates can only be loaded by someone sitting at console in a locked room under heavy guard, this may be acceptable.

TIS studied the safety of updates to a live DTE policy [23]. However, their work was aimed at updates intended only to *extend* a policy. Updates which violated current secrecy or integrity properties were refused. This was useful in their situation, where a project team would provide a policy update which increased control over objects (and subjects) which they already controlled. For instance, they might increase control over their own CVS root.

The updates which most people would like to apply would most likely violate existing security properties. For instance, while configuring a new login service, such as SRP [55], one might find one had forgotten to provide write access to the utmp file. A policy update to grant this access would most likely be refused by TIS' update system.

We believe refusing policy updates or removals is the responsible choice. For the case of testing a new service, as in the above example, or testing an entirely new policy, our DTE implementation provides a verbose mode which reports, but does not reject, access violations.

3.2.5 Networking

The TIS implementation of DTE assigned types to network packets [2]. DTE systems explicitly sent the type, as well as the domain of the originating process, along with network packets sent to other DTE systems. Network data from non-DTE systems are assigned types based upon the address of the originating host.

This DTE implementation does not mediate network access. On a secured network, between DTE systems whose policies are closely synchronized, the ability to have the DTE subsystem assign domain and type information, and mediate access accordingly, may be useful. In most cases, however, there are several reasons why this is not trustworthy. First, any discrepancy between the DTE policies on two machines can make it unsafe for one machine to trust the DTE information assigned to network data by the other machine. Second, provided the network is not secure, an insidious machine could impersonate a valid machine, in order to either observe or corrupt sensitive data. Clearly additional security measures can be added in order to make the DTE information on network packets. However, the resulting security does not justify the additional complexity at the operating system level.

DTE is better applied to network security by using DTE to protect encryption and signature keys on the local system, and using these keys to authenticate and encrypt network data.

3.3 Data Structures

Most data structures used by DTE are static, set up at the parsing of the configuration file. Two arrays are filled with null-delimited lists of names, one with names of types, and the other with pathnames. All structures which reference types or pathnames will point into these structures, reducing a great number of string comparisons to pointer comparisons.

The specification of each domain is represented in memory by a `dte_domain_t` structure. Every process' `task_struct` points to the `dte_domain_t` structure representing the domain under which it is running. The `dte_domain_t` structure contains lists representing all of the domain's access to types and domains. Access to a type is represented by a structure containing a bitmask indicating the type of access, and a pointer into the list of typenames. Storing domain access is more complicated. First we store the list of entry types, that is, the names of types which may be executed to enter this domain. Next we create a list of structures containing both a pointer to other domains, and an indication whether `auto` or `exec` access is allowed to the other domain. Third, we store a list of structures indicating which signals may be sent to processes running under other domains. Finally, in order to speed up the search for mandatory domain transitions, which must be performed on every file execution, we store an altered version of the domain transition list which we call the list of gateways. A gateway lists a domain name along with the name of one of its entry types. The list of gateways out of each domain contains an entry for each entry type to every domain to which the source domain has `auto` access. The gateways are stored on a hash list. In our prototype implementation, this setup allowed mandatory domain transition checks to be performed in constant time with respect to the policy size

and structure. While the structure has remained the same, the switching from entry point to entry type specifications will doubtless affect the running time. Since the prototype needed to recalculate the executable's pathname, whereas the current version can pick the typename from the inode, running time should be reduced even further.

Every inode structure contains a pointer to the name of the type assigned to the file. Consistent with the terminology presented in Section 3.2.2, the pointer to the inode's own type is called the *etype*, while the pointer to the default type for all its children is called the *utype*. As discussed in Section 3.2.2, the file's type can be decided in two ways.

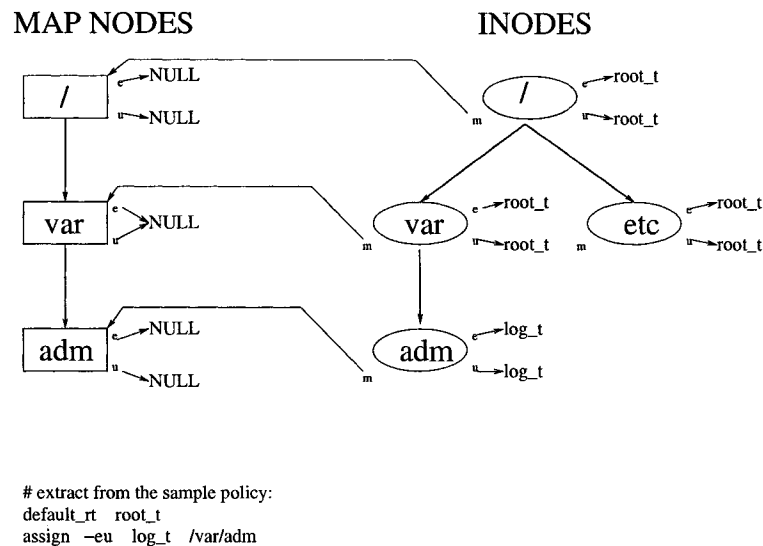


Figure 3.1: Inodes and corresponding mapnodes.

If the file is being created, or if there is no extended attributes file, then the policy's type assignment rules are used to assign a type to the inode. Type assignment rules are stored in memory by a tree of *mapnodes*. The mapnode tree structure mimics the file system tree. Figure 3.1 depicts a set of mapnodes alongside the associated inodes. When reading a mapnode for the first time, type assignment proceeds as follows. First we initialize both the

`etype` and `utype` to the parent inode's `utype`. Next we search for a mapnode corresponding to the current pathname. To find the mapnode corresponding to `/etc`, for instance, we look for a mapnode corresponding to the parent inode, and, if this exists, search its children for one named `etc`. If no mapnode corresponds to `/etc`, we retain the initialized values for the `utype` and `etype`. If a mapnode does exist, we link the inode to the mapnode for future reference, and overwrite the `etype` and/or `utype`, provided the values stored in the mapnode are non-NULL. Since we must represent the file system tree in mapnodes down to the deepest level specified in any type assignment rule, there will likely be mapnodes acting only as placeholders, not representing any type assignment rules themselves. In these cases, both `etype` and `utype` would be NULL.

Alternatively, an inode's DTE type can be read from an *extended attributes* file. The file pointer for this file is stored in the file system's `super_block`, along with the table relating typenames to indices for this file system. The file's `etype` is read from the extended attributes file. Unfortunately, we must compute and store the hierarchical type assignment information, that is, the `utype` and mapnodes, for inodes even if we used extended attributes to compute the `etype`. This is because we may create a file, or mount a file system, underneath this inode. In either case, we will need the inode's mapnode and `utype` information. This completely negates the performance advantage which extended attributes are supposed to provide. However, extended attributes remain useful for their resistance to the hard link, mounting, and namespace problems.

3.4 Algorithms

Following are some of the algorithms used by the current DTE implementation. The underlying design decisions are described above. The LSM functions are grouped according to the kernel functions to which they relate.

All DTE functions begin with a check for a variable called `dte_initialized`, which is set to true only when the security fields for all DTE-controlled kernel objects have been initialized. This is necessary because LSM binds the DTE functions before it gives DTE a chance to initialize the security fields on DTE-controlled kernel objects.

3.4.1 Mount

There are several mount-related LSM hooks. The first is initially called to ensure that the requested mount action is in fact allowed. DTE forbids mounting only if the policy file has specified a mount restriction for the device being mounted, as discussed in Section 3.2.2.

This condition is checked by the `dte_check_sb` function:

```
int dte_check_sb (struct vfsmount *mnt, struct nameidata *nd)
{
    struct super_block *sb = mnt->mnt_sb;
    struct dte_mntr *r;

    if (dte is initialized) {

        r = get_mount_restriction(sb->s_dev);
        if (r) {
            path = pathname of mountpoint;
            if (r->path != path && r is a strict mount) {
                return -EPERM;
            }
            if (r->path == path && r is a pretend mount) {
                // set "real" parent pointers;
                sb_sec->mnt_parent = mntget(nd->mnt);
                sb_sec->mountpoint = dget(nd->dentry);
            }
        }
    }
}
```

```

        }
    }
    return 0;
}

```

The `dte_check_sb` function also must perform a part of the hierarchical DTE information setup. Most of this is performed later, since we need information gathered during the remaining mount operation to complete it. However, in the case where the mounted device is associated with a pretend mount rule, but is being mounted in the specified pretend location, waiting until after the mount completes means that, when using the pretend location to specify the real parent `vfsmount` and mountpoint, the new device is already mounted on the pretend location. In this case, therefore, we must set up these pointers prior to the mount operation.

The `post_addmount` LSM function is called after the mounting of most file systems. The sole exception is the root file system, whose mount is followed by `post_mountroot`. As these two functions do much the same thing under DTE, we present only `post_addmount`.

```

void dte_post_addmount (struct vfsmount *mnt, struct nameidata *nd)
{
    /* set up external attributes file info */
    dte_setup_eafile(mnt->mnt_sb, mnt);

    /* set up hierarchical information */
    hierarchical_setup(mnt);
}

```

As discussed in Section 3.2.3, DTE handles file type resolution in two distinct ways. One is based upon the file system hierarchy. The other is based upon an external attributes file. Some initialization for each method is required at mount time. These are handled by the `hierarchical_setup` and `dte_setup_eafile` functions.

```

void dte_setup_eafile(struct super_block *sb, struct vfsmount *mnt)
{
    struct dte_sb_sec *sb_sec = sb->s_security;

    if (sb_sec->initialized)
        return; /* This device has previously been mounted */

    fp = open file "dteeaf" on this file system;
    sb_sec->ntypes = read number of types from fp;
    for (i=0; i<sb_sec->ntypes; i++) {
        sb_sec->type_conv[i] = read next type name from fp;
        sb_sec->initialized = 1;
        sb_sec->offset = location of start of typename list in fp;
    }
}

```

If the file has not been previously mounted, `dte_setup_eafile` checks for the existence of an extended attributes file. If this exists, then it builds an array binding integer indices to typenames, as specified by the file. It also records the offset of the start of the actual listing of extended attributes, but does not begin to read in types for any inodes. Finding the type for a particular inode number can now be done in constant time by adding the inode number, multiplied by the size of a type index in the extended attributes file, to the offset.

The hierarchical setup function also returns early if the file system has previously been mounted.

```

void hierarchical_setup(struct vfsmount *mnt)
{
    struct super_block *sb = mnt->mnt_sb;
    struct dte_sb_sec *sb_sec = sb->s_security;

    if (sb->initialized)
        return;
}

```

This function must set up the pretend mount location. This is done by setting two

pointers in the superblock's security field.² This function must also insert the inode for the root of the mounted file system into the mapnode tree (see Section 3.3). This is the information which is will be used to determine how the file system fits into the type assignment tree.

As mentioned above, if the device is mounted on its pretend location, then `dte_check_sb` will have set the pretend mount location on the superblock before mounting completes. In this case, we need only insert the inode into the mapnode tree.

```

    if (sb->mnt_parent has already been set up) {
        copy_dte_data(mnt->root_inode,
                    sb_sec->mountpoint->inode);
    }

```

If no pretend mount location is specified for this device, then we set the pretend location to the mount point. If this device is mounted a second time, then `hierarchical_setup` will shortcut, and the pretend mount location will continue to point to this first mount point.

```

    if (no pretend mount restrictions for this device) {
        sb_sec->mnt_parent = mnt->mnt_parent;
        sb_sec->mountpoint = mnt->mnt_mountpoint;
        copy_dte_data(mnt->root_inode,
                    sb_sec->mountpoint->inode);
    }

```

Finally, if the device is associated with a pretend location, but is being mounted elsewhere, then we set up the pretend location here. The reason we cannot also handle this case in `dte_check_sb` is that we must perform a path lookup to obtain the `(dentry, vfsmount)` pair corresponding to the pathname. We cannot do this in `dte_check_sb` because that

²Two pointers are needed because a pathname in the Linux kernel is uniquely identified by a `(dentry,vfsmount)` pair. In fact we are only after one thing, the pathname, but no such thing exists in the kernel.

is called holding a lock which may not be held for the path lookup function. The path lookup is not needed in `dte_check_sb` because we already have the mount point's (`dentry`, `vfsmount`) pair.

```

        else {
            path_lookup(pretend path name, &nd2);
            sb_sec->mnt_parent = nd2.mnt;
            sb_sec->mountpoint = nd2.dentry;
            copy_dte_data(mnt->root_inode,
                        sb_sec->mountpoint->inode);
        }
    }

```

3.4.2 File Type Resolution

File type resolution is accomplished using the mapnodes, as described in Section 3.3. By properly setting mapnode information on root inodes at mount time, we do not need to worry at all about the issues of multiple mounts or directory binding at inode lookup time. The following algorithm is implemented as `dte_real_postlookup`, which is called by LSM after any inode is first read from disk, that is, the first time a file is read.

The hierarchical type assignment information must be maintained even if we use extended attributes, in case a file system not using extended attributes is mounted at a lower level. Barring any contradictory type assignment rules, both the `etype` and `utype` are inherited from the parent directory's inode.

```

static inline void dte_real_postlookup (struct inode *ino,
                                       struct dentry *d, int create)
{
    struct dte_inode_sec *p, *c;

    /* assign types using the hierarchical scheme */
    c = inode->security;
    p = inode->parent->security;

```

```

c->utype = c->etype = p->utype;
c->initialized = 1;

c->map = mapnode_getkid(p->map, d->name);
if (c->map != NULL) {
    c->map = mapnode_getkid(p->map, d->name);
    c->etype = c->map->etype;
    c->utype = c->map->utype;
}

```

The function returns if the device does not have an external attributes file. It also returns if the inode is of a newly created file. This is because DTE uses type assignment rules to determine the type of files which are created on a file system using an external attributes file. This contrasts to SELinux' TE and TIS' DTE, which use the creating process' domain and the type of the parent directory to assign a type.

If the file is not new, and the file system uses external attributes, then the file's types are taken from the external attributes file:

```

if (create ||
    this device has no external attributes file) {
    return;
}

buf = read type index from ea file at offset+inode #;
c->utype = c->etype = convert buf to type name;
}

```

We now show informally how this algorithm, combined with the above mount algorithms, correctly handles multiple mounts and directory binding.

To show that we correctly handle multiple mounts, we will access a file `dir1/dir2/file`, which is located on a file system mounted at least twice. If the file system stores typenames as external attributes, then, as there is no ambiguity in inode numbers, the same type will be returned for `dir1/dir2/file` regardless of file system mount activity. We therefore

assume there is no external attributes file. The `dte_real_postlookup` function bases the types of files (and directories) based upon the mapnode tree information stored at the parent directory, using the following excerpted pseudo code:

```
c->map = mapnode_getkid(p->map, d->name);
if (c->map != NULL) {
    c->map = mapnode_getkid(p->map, d->name);
    c->etype = c->map->etype;
    c->utype = c->map->utype;
}
```

Therefore the types returned for `dir1/dir2` and `dir1/dir2/file` depend purely on the correct insertion of `dir1` into the mapnode tree. This is done at mount time.

The `dte_post_addmount` function calls `hierarchical_setup`, which first performs the following check:

```
if (sb->initialized)
    return;
```

In other words, if the superblock is already initialized, which means that this device has been previously mounted, then we do not continue. Therefore, the first path under which the device was mounted will continue to be that used for file type resolution, as it should be. The simple case of multiple mounts is therefore (trivially) correctly handled.

Later, `hierarchical_setup` contains the following code:

```
if (no pretend mount restrictions for this device) {
    ...
    clone security field on mnt's root inode from
        the mountpoint dentry->d_inode.
} else {
    path_lookup(pretend path name, &nd2);
    ...
    clone security field on mnt's root inode from
        nd2.dentry->d_inode;
}
```

Cloning an inode's security field also copies its pointer into the map node tree. If the DTE policy configuration file specifies a pretend mount location for this device, the mapnode for that location is used for the root inode of the newly mounted file system. Otherwise, the mapnode for the specified mountpoint is used.

Finally, looking again at the `dte_real_postlookup` function, when `dir1` is looked up, we will look at the parent inode's mapnode for a child mapnode by `dir1`'s name. If this does not exist, we copy the parent inode's `utype` to `dir1`'s `etype` and `utype`. Since the parent inode is the inode for the file system's root dentry, and since we have shown that this inode's security field is correctly assigned at mount time, multiple mounts are correctly handled by the file type assignment algorithm.

Directory binding is a remount of a specific directory within an already mounted file system, on top of a new mountpoint. The very act of reading the directory as part of binding it, ensures that the security field for the inode for this directory is already initialized. This is not a part of the DTE code, but occurs within the `do_loopback` function at the line

```
err = path_lookup(old_name, LOOKUP_FOLLOW, &old_nd);
```

It will therefore have been correctly set before it was ever bound. We refrain from explicitly constructing the obvious inductive proof.

3.4.3 Inode Permission

Since type names are set at the time when inodes are first read from disk, the DTE inode permission function is quite simple. It consists only of a few safety checks, followed by a check through the current domain's list of type accesses for the requested access to the inode's type. Checks for file execute permission are delayed until after a domain transition.


```

static inline int dte_real_inode_permission(struct inode *inode, int mask)
{
    struct dte_ta *ta; /* a dte type access structure ptr */

    ta = search for type access from current domain to inode->etype;

    if (!ta)
        return;

    if (S_ISDIR(inode->i_mode)) {
        if ((mask&MAY_EXEC) && !(dte_descend_access(ta->access)))
            DENY_ACCESS("dir x");
        if ((mask&MAY_WRITE) && !(dte_create_access(ta->access)))
            DENY_ACCESS("dir w");
        if ((mask&MAY_READ) && !(dte_readdir_access(ta->access)))
            DENY_ACCESS("dir r");
    } else {
        if ((mask&MAY_WRITE) && !(dte_fw_access(ta->access)))
            DENY_ACCESS("file w");
        if ((mask&MAY_READ) && !(dte_fr_access(ta->access)))
            DENY_ACCESS("file r");
    }
    return 0;
}

```

3.4.4 Execve

Section 2.5.6 details the two types of domain transitions allowed under DTE. These are called auto and exec transitions. A process must request an exec transition. This is done using a new system call:

```
static long dte_do_exec(void *data)
```

The pointer sent to this function must reference the following struct:

```

struct dte_exec_struct{
    char *fnam;
    char **argv;
    char **envp;
    char *domain;
};

```

The system call begins by reading this structure from user space. It searches the DTE exec rules for one permitting the current domain to transition to `data->domain`.

```

    ts = current->security;
    if (!dte_may_exec_to(ts->dte_domain, dest)) {
        log("dte: domain %s may not exec to domain %s.\n",
            ts->dte_domain->name, dest->name);
        err = -EACCES;
        goto out_putf;
    }

```

We next back up the current domain, set the new domain, and begin execution of the requested file using the standard `execve` system call. We must temporarily store the old domain for three reasons. First, since the executable file `data->fnam` may not yet have been loaded from disk, we wait until we are certain the file's type has been calculated in order to determine whether this file is an entry point to the destination domain. If it is not, then the process must be returned to its original domain. Similarly, if execution fails early on, we must also reset the domain. An example of such a failure would be the file `data->fnam` not existing on the file system. Third, since an auto transition overrides an exec transition, we will need the old domain in order to ensure that no auto transition existed for the old domain and the new executable file.

Finally, if the `execve` system call fails, we reset the original domain.

```

    ts->dte_back = ts->dte_domain;
    ts->dte_domain = dest;
    err = execve(es.fnam, es.argv, es.envp);
    if (ts->dte_back) {
        ts->dte_domain = ts->dte_back;
        ts->dte_back = NULL;
    }

```

The `execve` system call calls the function `prepare_binprm`, which calls the LSM function `bprm_set_security` in order to set security module information. In DTE, this function

handles both auto domain transitions, and the final entry point check for exec transitions. It begins by checking for an auto transition, using the `dte_auto_switch` function. If such a transition is required, it takes place, potentially overriding a requested exec transition. Otherwise, if an exec transition has been requested, we ensure that the file being executed is an entry point to the new domain.

```

        if (!dte_auto_switch(s->etype)) {
/* Log the mandatory domain switch */
        } else if (ts->dte_back &&
                !dte_domain_has_ep(ts->dte_domain, s->etype)) {
/* this domain transition is not allowed */
        log("dte: type %s is not ep to domain %s.\n",
            s->etype, ts->dte_domain->name);
        ts->dte_domain = ts->dte_back;
        ts->dte_back = NULL;
        return -EACCES;
    }

```

Next execution is attempted. If it fails, the process' original domain is reset. Otherwise, the process begins execution of the new file under its new domain.

```

    ret = dte_check_x(ts->dte_domain, s->etype);
    if (ret) {
        /* not allowed */
        log("dte: domain %s may not execute type %s.\n",
            ts->dte_domain->name, s->etype);
        if (ts->dte_back) {
            ts->dte_domain = ts->dte_back;
            ts->dte_back = NULL;
        }
        return ret;
    }
    ts->dte_back = NULL;
    return 0;
}

```

Section 3.3 explains that a list of gateways is created for each domain. The function `dte_auto_switch`, called from the DTE version of `bprm_set_security`, searches the list

of the source domain's entry types for the type assigned to the file being executed. If it is found, then `dte_auto_switch` returns 0, indicating that an auto domain transition is required. Otherwise, it returns 1. Since the check for an auto transition is reduced to one hash calculation, performance impact is minimized despite `dte_auto_switch` being called for every file execution.

```
static int dte_auto_switch(char *type)
{
    ts = current->security;
    curd = ts->dte_domain
    if (ts->dte_back) {
        /* in a dte_exec, we want to use the original domain */
        curd = ts->dte_back;
    }
    if (curd has no gateways)
        return 1;
    gw = entry in gateway hash table for type;
    if (gw) {
        if (ts->dte_back) {
            log overridden exec transition;
            ts->dte_domain = gw->domain;
        } else {
            ts->dte_back = ts->dte_domain;
            ts->dte_domain = gw->domain;
        }
        return 0;
    }
    return 1;
}
```

3.4.5 DTE Module Init

The switch to LSM involved allowing DTE to be installed as a module after system boot had completed. Important kernel structures have already been created, without DTE security information attached, and important information has already been lost. For instance, while we have access to the process tree, processes may have already reparented themselves. More

importantly, for processes which have performed sequences of `execve`, only the filename of the latest `execve` invocation is available. This means that we cannot know how to assign domains to processes correctly. To cope with this situation as best we can, we use the following algorithm:

```
int setup_dte_module(void)
{
    /* read dte config file ... */

    /* assign a specified default domain to all processes: */
    lock_kernel();
    for_each_task(taskp) {
        taskp->security = kmalloc(sizeof(struct dte_task_sec),
            GFP_KERNEL);
        task_sec = (struct dte_task_sec *)taskp->security;
        task_sec->dte_domain = default_domain;
        task_sec->dte_back = NULL;
    }

    /* Set up the root of the file system... */

    /* And walk the entire file system tree loaded so far, assigning
       DTE types: */
    dte_walk_dcache_tree_full(root_mnt, root_sb->s_root);
}
```

3.5 Configuration File

When a DTE system boots, it reads a policy from the configuration file, which is located at `/.dte/dte.conf`. This file is structured as follows. It begins with a declaration of the types and domains used in this policy. Next it defines the default domain, which is assigned to the first process. It also defines the `etype` and `utype` for the root of the file system. This is followed by domain specifications and type assignment rules.

```

<dte_config> ::=
    types <type_name> { <type_name> }
    domains <domain_name> { <domain_name> }
    default_d <domain_name>
    default_et <type_name>
    default_ut <type_name>
    <domain_spec>
    { <domain_spec> }
    <type_assignment>
    { <type_assignment> }

```

Each domain specification lists, in order, the entry types, the domain's access to types, its permitted and required transitions to other domains, and the signals which it may send to other domains. Each domain access may be auto or exec. Type access may be any combination of "r" for file read, "w" for file write, "x" for file execute, "l" for directory read (lookup), "c" for directory write (create), "d" for directory descend, or "a" for file append. Enforcement of file append is, however, not yet implemented. The signal in a signal access rule may be a comma-delimited list of signal numbers, or "0" for "any signal."

The list of type assignments follows. A type assignment statement may associate several paths with one type. Each statement binds the pathnames as "-e" for explicit, "-u" for under, or "-eu" or "-r" for both. Their meanings are discussed in Section 3.2.2.

```

<domain_spec> ::= spec_domain <domain_name>
    (<number> { <entry_type> } )
    (<number> { <type_access> } )
    (<number> { <domain_access> } )
    (<number> { <signal_access> } )
<type_assignment> ::= assign <assign_option> <type_name> <path_name>
    { <path_name> }

```

Each of the above statements must adhere to a very specific syntax. Their specifications follow.

```

<assign_option> ::= "-e"|" -u"|" -r"|" -eu"

<entry_type> ::= <type_name>
<type_access> ::= <type_acc_token> "->" <type_name>
<domain_access> ::= <domain_acc_token> "->" <domain_name>
<signal_access> ::= <number> "->" "0"
<signal_access> ::= <number> "->" <domain_name>
<name> ::= <letter> { <letter> | <digit> }
<type_name> ::= <name>
<domain_name> ::= <name>
<path_char> ::= <letter> | <digit>
<path_element> ::= <path_char> { <path_char> }
<path_name> ::= "/" <path_element> { <path_name> }
<number> ::= <digit> { <digit> }
<type_acc_token> ::= <type_acc_piece> { <type_acc_piece> }
<type_acc_piece> ::= "r"|"w"|"x"|"l"|"c"|"d"|"a"
<domain_acc_token> ::= "auto" | "exec"

```

A few notes follow which are not expressed in the BNF specification. First, to continue any statement onto the following line, the line must be ended with a \. Second, in domain specifications, the numbers preceding each of the type, domain and signal accesses must be the exact number of access specifications following.

3.6 DTE API

DTE multiplexes the LSM `security` system call to provide three ways of interacting with the DTE subsystem. Since these three are provided through one system call, they are not themselves true system calls. However, as they allow user-space programs to interact with the kernel through a kernel trap, and since they most closely resemble system calls in purpose and function, we will nevertheless refer to them as syscalls.

The three DTE syscalls are called `get_type`, `get_domain`, and `dte_exec`. They are invoked by calling the `security` system call and sending the integer code 10 as the first

argument, in order to specify that these are directed toward the DTE subsystem; an integer between one and three as the second argument to identify the specify DTE syscall; and a pointer to a structure as the third argument. The structure in the third argument contains the actual arguments being used by the DTE syscall.

The code snippets below explain the usage of these syscalls.

- `get_type`

This call is used to learn the type assigned to a file. It receives a string containing the pathname to be queried, as well as a buffer and an integer specifying the buffer size. The type name is stored in the buffer.

```
struct dte_gt_struct {
    char *fnam;
    char *buf;
    int buflen;
};
int security(int id=10, int call=1, struct dte_gt_struct *gt);
```

- `get_domain`

This call is used to learn the domain under which a process is running. It receives an unsigned integer representing the process id, as well as a buffer and an integer specifying the buffer size. The domain name is stored in the buffer.

```
struct dte_gd_struct {
    unsigned int pid;
    char *buf;
    int buflen;
};
int security(int id=10, int call=2, struct dte_gd_struct *gd);
```


- `dte_exec`

This call requests a voluntary transition to a new domain while beginning execution of a new file. It is sent the pathname to be executed, the domain to which to transition, as well as the lists of arguments and environment variables. If execution of `fnam` triggers a mandatory domain transition, then execution proceeds under the required domain, and the requested domain transition does not occur.

```
struct dte_exec_struct{
    char *fnam;
    char **argv;
    char **envp;
    char *domain;
};
int security(int id=10, int call=3, struct dte_exec_struct *de);
```

3.7 Effectiveness

To show the effectiveness of our DTE implementation, we picked a high-profile vulnerability, namely the buffer overflow in *wu-ftpd*[49], and showed how our implementation of DTE can prevent an attacker from obtaining a root shell. Our goal was to show that we could protect the system from the *wu-ftpd* vulnerability (the posted exploits as well as future or hand-crafted ones) without modifying the binary. In order for ftp to retain its full functionality, it would need to be made DTE-aware so that it could, like login, allow ftp to transition into the domain associated with a user being authenticated. We did not do this, but set protections such that users can retrieve files from, if not deposit files onto, the server. Anonymous ftp was fully functional.

The policy shown in Figure 3.2 prevents domain *ftpd_d* from executing any system binaries other than `/usr/sbin/in.ftpd` and binaries located under `~ftp/bin/` (lines 19-21). These files are defined to be of the type `ftpd_xt` (lines 29 and 30), which the domain *ftpd_d* may execute but not write (line 20). Only *ftpd_d* may execute this type (lines 9-21), and *root_d* automatically switches to *ftpd_d* on execution of `/usr/sbin/in.ftpd` (line 12), since that is an entry point to *ftpd_d* (line 19). The exploits to be found on the internet to take advantage of this vulnerability will therefore fail, as they expect to be allowed to run `/bin/sh`. Nor can a script be written to upload and run a Trojan horse, since the only types which *ftpd_d* is allowed to write may not be executed by anyone.

The script which we tested was `wuftpd2600`, which can be found at the Security Focus website [20]. It connected to our test machine, and exploited the buffer overflow. However, the DTE-enabled kernel refused to allow the *ftpd_d* domain to execute `/bin/sh`. The script therefore hung, and the system was not compromised. The error messages in Figure 3.3 were sent to *syslog*. In contrast, the plain 2.3.28 kernel happily provided a root shell.

```

01 # ftpd protection policy
02 types root_t login_t user_t spool_t binary_t lib_t passwd_t shadow_t dev_t \
03     config_t ftpd_t ftpd_xt w_t
04 domains root_d login_d user_d ftpd_d
05 default_d root_d
06 default_et root_t
07 default_ut root_t
08 default_rt root_t
09 spec_domain root_d (/bin/bash /sbin/init /bin/su) (rwxcd->root_t \
10     rwxcd->spool_t rwcx->user_t rwdc->ftpd_t rxd->lib_t rxd->binary_t \
11     rwxcd->passwd_t rwxcd->shadow_t rwxcd->dev_t rwxcd->config_t \
12     rwxcd->w_t) (auto->login_d auto->ftpd_d) (0->0)
13 spec_domain login_d (/bin/login /bin/login.dte) (rxd->root_t rwxcd->spool_t \
14     rxd->lib_t rxd->binary_t rwxcd->passwd_t rwxcd->shadow_t rwxcd->dev_t \
15     rxwd->config_t rwxcd->w_t) (exec->root_d exec->user_d) (14->0 17->0)
16 spec_domain user_d (/bin/bash /bin/tcsh) (rwxcd->user_t rwxcd->shadow_t \
17     rwxcd->spool_t rxd->lib_t rxd->binary_t rwxcd->passwd_t rxwd->root_t \
18     rwxcd->dev_t rxd->config_t rwxcd->w_t) (exec->root_d) (14->0 17->0)
19 spec_domain ftpd_d (/usr/sbin/in.ftpd) (rwdc->ftpd_t rd->user_t rd->root_t \
20     rxd->lib_t r->passwd_t r->shadow_t rwdc->dev_t rdx->ftpd_xt \
21     rd->config_t rwdc->w_t d->spool_t) () (14->root_d 17->root_d)
22 assign -u /home user_t
23 assign -u /tmp spool_t
24 assign -u /var spool_t
25 assign -u /dev dev_t
26 assign -u /scratch user_t
27 assign -r /usr/src/linux user_t
28 assign -u /usr/sbin binary_t
29 assign -e /usr/sbin/in.ftpd ftpd_xt
30 assign -r /home/ftp/bin ftpd_xt
31 assign -e /var/run/ftp.pids-all ftpd_t
32 assign -r /home/ftp ftpd_t
33 assign -e /var/log/xferlog ftpd_t
34 assign -r /lib lib_t
35 assign -e /etc/passwd passwd_t
36 assign -e /etc/shadow shadow_t
37 assign -e /var/log/wtmp w_t
38 assign -e /var/run/utmp w_t
39 assign -u /etc config_t

```

Figure 3.2: A DTE policy to protect from *wu-ftpd*, with line numbers added.

```

Aug  4 13:12:03 wicked kernel: do_exec: d_t_check_x returned 1(exec denied).
Aug  4 13:12:03 wicked kernel: do_exec: domain ftpd_d type root_t.

```

Figure 3.3: Error messages resulting from attempted *wu-ftpd* exploit.

Chapter 4

Performance

We began the performance analysis of LSM and DTE by using the LMBench [36] benchmark suite. Some of the LMBench benchmark attempts to measure hardware performance, and therefore was not relevant. We present and discuss the relevant LMBench results in Section 4.1. For those areas which the benchmark showed were significantly affected by DTE, we analyze the cause of the performance impact in more detail in Section 4.2 by directly timing the kernel operations involved. That section also investigates some performance hits which we expect, but which LMBench does not measure. Finally, in Section 4.3 we perform a macro-benchmark to analyze the overhead perceived by users of the system.

All benchmarks were performed on a 400 MHz Pentium II class system with a 2.51 nanosecond clock and 128M ram. LMBench calculated the actual clock speed as 398MHz (1/2.51). Therefore, for all tests which measured clock cycles, we report results in microseconds calculated by dividing by 398. Background processes were kept to a minimum by not starting services such as X windows, lpd and cron.

4.1 LMBench Results

The LMBench suite was executed ten times on each of an unaltered 2.5.6 kernel, an LSM-enabled 2.5.6 kernel with no modules loaded, an LSM kernel using only the capabilities module, an LSM kernel with the DTE module loaded, and an LSM kernel with the capabilities module stacked on top of the DTE module. The means and standard deviations are presented in Appendix B. In our discussion, we will mainly compare the stock 2.5.6 kernel, the LSM kernel with the capabilities module loaded, and the LSM kernel with the DTE module loaded. The capabilities module rarely made a significant impact on the results. However, we use it rather than LSM with no modules, because the stock kernel uses capabilities. Whereas capabilities significantly enhance the security of the stock kernel, the DTE kernel can be made secure without the aid of the capabilities module. We therefore consider the most appropriate comparisons to be of the stock kernel, the LSM kernel using the capabilities module, and the LSM kernel using only the DTE module. Section 4.1.1 presents file system and virtual memory performance. Section 4.1.2 presents process-related performance. Since the context switch and memory latency results are strictly hardware measurements, we do not discuss them here.

4.1.1 File System and VM Performance

4.1.1.1 Mmap

Counter-intuitively, mmap latency improved in LSM, and improved even more for the DTE module. The difference amounted to less than 0.5%. Nevertheless, mmap is an important method of file access, and the lack of performance impact due to either LSM or DTE is a

pleasing, if expected, result.

4.1.1.2 File Creation

File creation involves the following LSM hooks from `inode_ops`: `create()`, `post_create()`, and a call to `permission()` for each parent directory contained within the pathname. In DTE, the `create()` call is empty, while the `post_create()` call performs the same tasks which would be needed if the file were read from disk for the first time. This includes determining the appropriate DTE types and, if necessary, hooking into the mapnode tree. If extended attributes are in use, then `post_create()` must also write the newly determined type for this inode into the extended attributes file.

The LSM code introduced a 2-3 microsecond overhead for creation of any size file. The DTE code introduced an additional six to eight microseconds.

4.1.1.3 File Deletion

File deletion involves the following LSM hooks: `inode_ops→permission()` for EXEC permission at the parent inode, followed by `inode_ops→unlink()`. The `unlink()` function is empty for DTE. Any extra cost incurred by DTE over LSM is, therefore, the same as if we were simply accessing the file. Figure 4.1 shows the relevant measurements for the file deletion benchmarks. The first column shows the size of files being deleted, the second column shows the deletions per second, and the third column shows the time (in microseconds) to delete a file. Note that the similar times for 1k and 4k file deletions are accounted for by the file system's 4k blocksize. The DTE overhead appears to be about 2.9 microseconds above the LSM time.

At this point a note about precision is in order. While the 95% confidence interval for 10 runs of our LMBench file deletion results was 0.15 microsecond, a subsequent trial, on the same hardware but a rebuilt system, returned numbers which differed by as much as 4 microseconds from the first run, but again exhibited a 95% confidence interval of 0.15 microsecond. This suggests the possibility that the overhead depends greatly upon particular conditions in the file system's free inode and free block bitmaps.

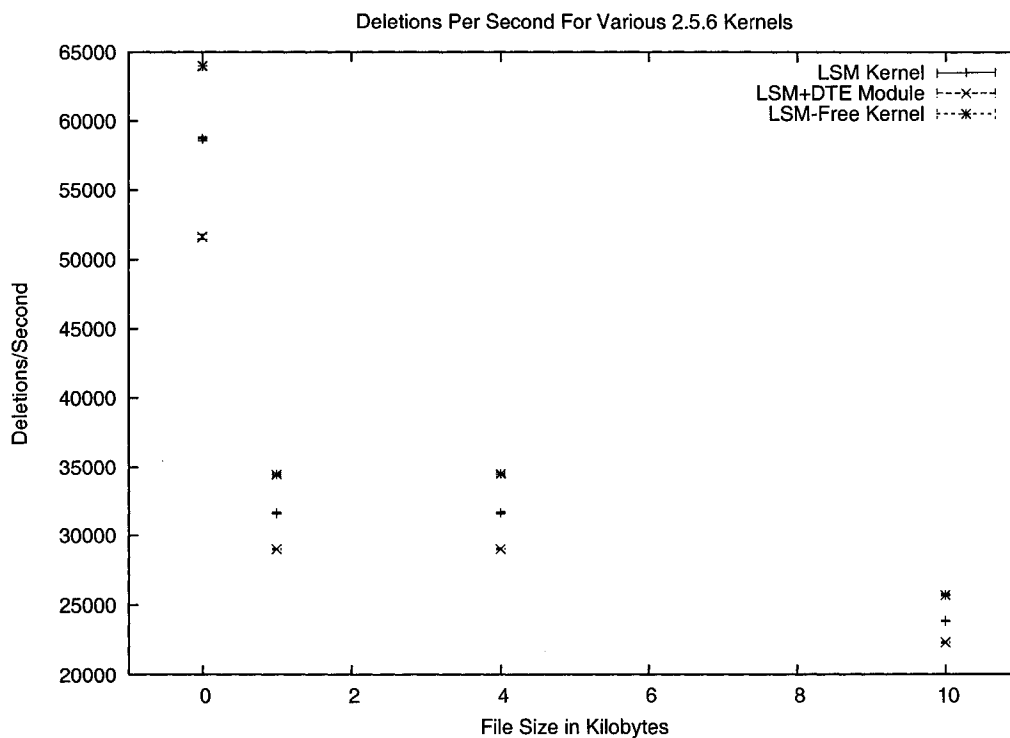


Figure 4.1: LMBench results for file deletion

4.1.2 Process-Related Performance

4.1.2.1 Null Call

A system call allows user code to interact with the operating system. To obtain a measurement of the time to perform a null system call, that is, the time required for only the

operating system overhead involved in performing a system call, LMBench uses a call to `sys_getppid`. This is a short system call, into which LSM fortunately inserts no hooks. Therefore it remains a good null system call test for us. It is mentioned here only because of the unexpected result of the LSM kernel outperforming the plain kernel, with all combinations of inserted modules. This would appear to be an artifact of the extra LSM information in memory, purely by accident, resulting in better distribution of data in cache.

4.1.2.2 Stat() and Open()/Close()

For our purposes, these benchmarks are closely related. This is because both `open()` and `stat()` call `open_namei()`, which calls `permission()` and `post_lookup()` for each path element leading up to the filename. Provided the file's dentry information is cached, as it is when the same file is repeatedly opened and closed, no other LSM hooks are called.

The DTE overhead is significant, adding 33% to the stock kernel's time for `stat()` and 29% for `open()/close()`, as opposed to the 4% increase for `stat()` and 2% increase for `open()/close()` imposed by the capabilities module. The DTE `permission()` and `post_lookup` functions are quite short, and do not seem to account for this overhead. We therefore analyze this further in section 4.2.1.

4.1.2.3 Signals

The time to install a signal handler was not affected by either LSM or DTE. However, the time to send a signal, as expected, was affected. The capabilities module introduced 0.1 microsecond overhead over the stock kernel, and the DTE module added an additional microsecond. This represents the cost of a single LSM function call, which involves two

pointer dereferences, the saving of the return value, and the following test of the return value. The hook being called, for both the dummy and capabilities module, does only a

```
return 0;
```

The benchmark sends a signal only to itself. Since DTE shortcuts for any signal sent to a process within the same domain, this is not a good test of the DTE signal code. Therefore, micro-benchmarks to further investigate DTE signal performance are developed in Section 4.2.3.

4.1.2.4 Fork

The time to fork a new process did not differ significantly between kernels. Indeed, forking and exiting a process introduce only three LSM hooks, each of which consists of simply

```
return 0;
```

for the dummy and capability modules. In DTE, the overhead is constant, always consisting of one memory allocation, two conditionals, and four simple assignment statements.

4.1.2.5 Fork and Exec

The addition of an `exec()` call introduces a much larger amount of work for the DTE module. The kernel must now check the DTE policy for mandatory domain transitions, which will depend upon the current domain and the type of the file being executed. The DTE module did not perform significantly worse than the DTE-free LSM kernel. This result confirms the validity of using a hash table of gateways (see Section 3.3) to speed up the search for required automatic domain transitions. Under the DTE policy used during

these tests, the `user_d` domain, under which the benchmarks were run, had auto transition rights to 4 domains, through 4 gateways. However, a poor hash table implementation could cause the cost of checking for domain transitions to increase as a function of the number of gateways. Therefore, Section 4.2.2 will analyze this cost in more detail.

4.2 Micro Benchmarks

LMBench is designed as a general benchmark to test OS and hardware performance. We now present more detailed tests of some parts of the DTE code. We chose to profile any code involved in suspicious or disappointing results from LMBench, as well as any code which we felt was not adequately profiled by LMBench.

4.2.1 `Permission()`

As mentioned in the LMBench results, the `dte_inode_permission()` function appears to take an inordinate amount of time. This function only calculates the hash value of a type name, steps through the list of hash collisions to find the requested type name, and performs a few comparisons to determine whether to grant access. An obvious potential bottleneck is therefore the hash function, which we investigate first.

The hash function we use is that used by the Linux directory cache, or `dcache`. The `dcache` takes pathnames, descends the directory tree, and returns a file's inode number. Its hash table has therefore been thoroughly examined [32] and optimized. However, in order to minimize memory usage while accommodating for the fact that the number of hash entries will vary, we simply used a hash table of the exact size needed to store the number of entries. To measure the impact of this memory optimization on performance, we

calculated the maximum depth of any domain to type access specification in the chain of hash collisions, for each domain specification. In our stock kernel, the mean maximum over all domains was 6.2 ± 2.6 . By switching to hash tables containing twice the needed number of entries, we reduced the average of maxima to 5.1 ± 2.6 . It appeared worthwhile to make this same change to the hash tables storing entry points, signal accesses, domain accesses, and gateways, and investigate the resulting performance impact. However, doing so gave mixed results, as some LMBench results grew worse.

Our next step was to time the `open_namei()` call, the `dte_inode_permission()` function as a whole, and two pieces of the `dte_inode_permission()` function. In particular, we timed the actual calculation of the hash value of the type name, and the subsequent search through the list of collisions. Outliers¹ were removed from the data sets, as these generally reflected disk reads, which are far slower than the action being timed, and render data meaningless by making the (already large) standard deviation far larger than the mean. Our results come from three separate runs. The first timed `open_namei()`, the second timed `dte_inode_permission()`, and the third timed the hash operations. We ran the LMBench file system latency test each time. The separate runs were necessary since `open_namei()` calls `permission()`, and `permission()` contains the hash operations, so that the action of logging deeper timing results would affect the calling functions.

The DTE kernel's `dte_inode_permission()` function took an average of $.64 \pm .005$ microseconds. The permission function mainly calculates the hash value of the typename, and searches through hash collisions for the correct hash table entry. Calculating the hash

¹We defined outliers as those numbers which were at least an order of magnitude larger than the numbers which were not outliers. A significant gap existed between those numbers which we removed, and those which we retained.

value took $.367 \pm .002$ microseconds, while stepping through collisions took $.21 \pm .0005$ microseconds.

The file which was being accessed by LMBench during the `open()/close()` test was `/usr/tmp/lmbench`. This causes `permission()` to be called 4 times, accounting for 2.4 of the 3.3 microseconds of overhead in itself. In addition to this, we must add the function call overhead, as well as the overhead for `post_lookup`. Although each of these is small in itself, we have at least accounted for the majority of the overhead, which appears to stem from the actual hash calculation.

We also did a simple timing of all calls to `open_name()`. However, as this function is called for all pathname accesses, ranging from quick reads from cache, to reads from disk, to file creations, the numbers varied far too much to be of any use in comparing the small differences arising from the LSM and DTE performance hits. Severe outliers, which were an order of magnitude greater than the majority, were again removed, but variance was still too great for the numbers to be very informative. The `open_namei()` function averaged $125.85 \pm .96$ microseconds under a plain LSM-free kernel, and $134.31 \pm .98$ microseconds under DTE. Many calls, however, completed in less than 7.5 microseconds. Clearly, in order to garner meaningful information about the DTE overhead, we would have to use a more intrusive method of timing `open_namei()` to allow us to choose the instances timed. This intrusion, of course, would itself affect the results. Since we have measured the parts of `open_namei()` which will individually constitute DTE overhead, we did not perform further profiling.

4.2.2 Execve()

On file execution, the DTE kernel must search for a rule stating that the current domain must enter a new domain upon executing the new file. As this search was feared to become a tremendous bottleneck to DTE performance, the auto domain transition information is stored on a hash table for each domain. Therefore an issue which merits investigation is whether the amount of DTE overhead for file execution is constant, as should be the case with a hash table, or whether it grows as a function of the number of gateways out of the current domain. We created 10 domains, containing an increasing number of gateways from 0 to 100 in steps of 10. We then created a directory containing 100 differently named versions of *hello world*, none of which were actually gateways. In order to prevent console output from skewing our results, we closed standard output before executing *hello world*. Under a modified kernel which reports the run-time for `fs/exec.c:execve()`, we entered each of these domains, and executed a script which ran each of the 100 programs 10 times. Note that this is a very artificial test meant to find bugs or suboptimal code. In reality, the use of entry types, as opposed to entry points, means that few domains will ever need more than two entry types.

Table 4.2 lists the mean execution time and standard deviation for the execution times. Clearly, the number of gateways does not affect execution time. Combined with the LM-Bench results showing little overall performance impact of DTE on file execution times, this proves the efficacy of our design.

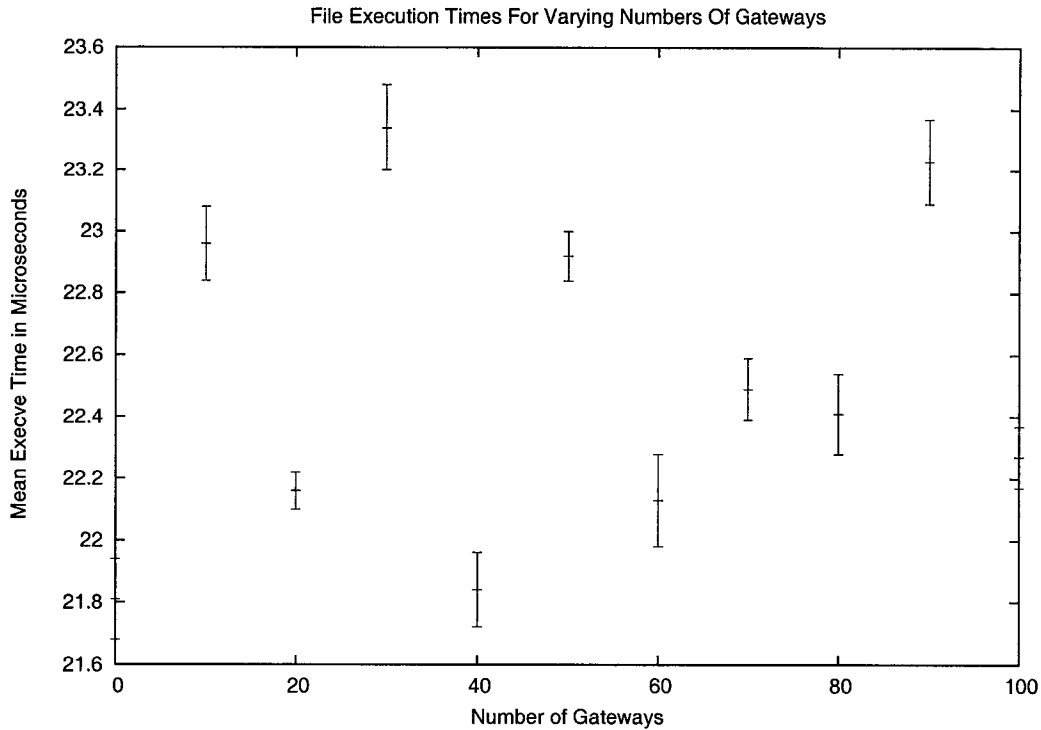


Figure 4.2: File execution times for varying numbers of gateways.

4.2.3 Signal

DTE only controls signals which are sent to processes running under a different domain than the process sending the signal. Therefore, the DTE function controlling signals shortcuts for signals which do not cross domain boundaries. The LMBench results measured only this shortened code path. We inserted profiling into the DTE signaling code in order to obtain more satisfying measurements. For signals which are not sent to a different domain, our tests measured an overhead of $.86 \pm .07$ microseconds. This is a little less than the LMBench results. However, as it does not include the time for a context switch while calling the DTE signal hook, the numbers do appear to agree. For signals crossing domain boundaries, DTE takes $2.40 \pm .21$ microseconds.

4.3 Macro Benchmark

All of the above benchmarking and micro-benchmarking is very useful in finding coding errors or code paths which may benefit from optimization. However it is far less reliable as an indicator of the total impact of DTE (and LSM) on real computational tasks. We therefore finish our performance evaluation with the commonly accepted standard for macro-benchmarking, namely a kernel compile. Under each of a plain 2.5.6 kernel, an LSM kernel with the capability module loaded, and an LSM kernel with the DTE module loaded, we perform

```
make clean
```

```
make bzImage
```

We timed only the actual compilation, not the "make clean". The first compilation under every kernel took 1007 seconds. However, the test machine had sufficient memory to keep all source code in memory after the first compile, so that new file creation and deletion became much more important. Under the plain kernel, subsequent compilations took 995 seconds. Under the LSM kernel with capabilities loaded, these took 996 seconds. Using the DTE module, they took 997 seconds. A performance impact of 0.1% certainly seems negligible in return for a robust and flexible MAC system.

Chapter 5

Access Rights of Domains

A feature of DTE systems which is too often ignored in analysis is the ability of entry points to restrict a domain's access rights. Here we attempt to gain a greater understanding of the effects which an entry point can have on a domain's real access rights. We begin with a few simple definitions.

Definition 5.0.1 *$RW(d)$ represents domain d 's immediate type accesses under some DTE policy.*

Note that $RW(d)$ is immediately available from a DTE policy file.

Definition 5.0.2 *$\mathcal{T}(d)$ represents the set of domains to which d is allowed to transition.*

Whether the transition is `auto` or `exec` is irrelevant to $\mathcal{T}(d)$. The purpose of the `auto` transition right is to accommodate legacy applications which are not DTE-aware. It is a useful architectural feature, but can be ignored here. Note that, if a file f_1 leads to an `auto` transition from one domain d_1 to another domain d_2 , the same functionality could be coded into another file f_2 of a different type, and executed under domain d_1 . Therefore `auto` transitions do not limit the access rights of d_1 , except in the rare case wherein d_1 cannot write or replace any files which it can execute. In any case $\mathcal{T}(d)$ is not affected.

Definition 5.0.3 $\mathcal{A}(d)$ represents the full set of type accesses which a process under domain d can exploit.

We say that a process can exploit a type access if it is able to execute arbitrary instructions taking advantage of this type access. For instance, to exploit the type access $r \rightarrow x_t$, a process must be able to read any files of type x_t at any time. Obviously, a process under domain d_1 can exploit $RW(d_1)$. If a process under d_1 is subverted, then the attacker can execute arbitrary code taking full advantage of any type accesses in $RW(d_1)$. If d_1 can transition to d_2 , the same process may or may not be able to exploit $RW(d_2)$. Whether or not it can will depend upon the entry points to d_2 . The same is true for $RW(d_3)$, where $d_3 \in \mathcal{T}(d_2)$. If all domains under some system have a shell as an entry point, then the full set of access rights of any domain is:

Axiom 5.1

$$\mathcal{A}(d_1) = RW(d_1) \cup (\forall d_i \in \mathcal{T}(d_1) : \mathcal{A}(d_i))$$

As an example, consider a policy wherein $d_3 \in \mathcal{T}(d_2)$, $d_2 \in \mathcal{T}(d_1)$, $ta_1 \notin RW(d_1)$, $ta_1 \notin RW(d_2)$, but $ta_1 \in RW(d_3)$. In this case, a process under d_1 could execute the following call:

```
transition(d2, "/bin/sh -c \"transition d3 /bin/sh -c \\\"rm f1\\\"")
```

This call would cause a domain transition to domain d_2 on execution of a shell. The arguments provided to this shell would in turn request a transition to domain d_3 on execution of another shell. That shell, in turn, could request removal of a file whose removal requires

type access ta_1 . In this way, a process running under domain d_1 can, at any time, exploit $RW(d_3)$.

We can prevent this by using stricter entry points. If d_2 can be entered only through a program which can only return the current time, and which cannot itself be subverted, then a process running under d_1 cannot force d_2 to execute a shell under d_3 with arbitrary arguments.

In order for a process under d to fully exploit $RW(d_n)$ where d_n is a domain to which d can transition, one of the following must be true:

- d can overwrite the entry point to d_n
- d can exploit a security vulnerability of an entry point to d_n
- The entry point to d_n allows arbitrary instructions to be executed. For instance, it is a shell.

We now begin to address the problem at hand:

Definition 5.0.4 $A_L(d)$ represents $A(d)$ as limited by the entry point of d .

In order to *safely* take into account the effect of entry points on a domain's access rights, we must ensure two things:

1. All entry points must not be writable or movable by anyone. If anyone is able to overwrite or replace an entry point, then we can no longer count on this entry point's restrictions to limit a domain's access.

This can be verified automatically by a generic policy analysis tool, as it is simply a feature of the DTE policy itself.

2. The code for all entry points must be verified to perform correctly. It takes only a single buffer overflow against an entry point to render all its protection useless.

This must be performed on a case by case basis. Henceforth, when we state an entry point's functionality, we assume that this verification has been correctly performed.

Clearly it pays to have a small number (preferably one) of entry points, and to make this (these) as simple as possible, so as to ease its verification.

We need a way of expressing the effect of an entry point on a domain's rights. However, an entry point will take into account some information which we can only glean from the actual process, such as a password offered by a user.

In the following, P represents a user process. We treat it as an object, dereferenced using “.”. For instance, $P.d$ returns the domain label for process P , while $P.pwd$ is interpreted as a password provided by the process and, presumably, by a user to the process. In the interest of brevity, we allow $\mathcal{A}(P)$ to be used as shorthand for $\mathcal{A}(P.d)$, $\mathcal{T}(P)$ for $\mathcal{T}(P.d)$, and $RW(P)$ for $RW(P.d)$. Whereas $\mathcal{T}(d)$ returns a list of domains to which d may transition, $\mathcal{T}(P)$ returns a list $C_j = (C_1, \dots, C_n)$ of processes, one for each domain to which $P.d$ may transition.

Definition 5.0.5 $d.\mathcal{L}$ is a function representing the entry point to d . It behaves as follows:

- $d.\mathcal{L}(RW(P))$ returns a subset of $RW(P)$, containing all elements of $RW(d)$ which the entry point to d , given the information stored under P , does not expressly forbid.
- $d.\mathcal{L}(\mathcal{T}(P))$ returns a subset of $\mathcal{T}(P)$, containing all elements of $\mathcal{T}(P)$ which the entry point to d , given the information stored under P , does not expressly forbid.

Finally, before we give a formula representing the access rights of a domain as limited by its entry points, we wish to rephrase a caution mentioned above more formally.

Axiom 5.2 *If a domain d has more than one entry point, or the entry point may be overwritten or replaced by any other domain, then*

$$\mathcal{A}_L(d) = \mathcal{A}(d).$$

That is, the entry points may not be assumed to limit the access of d . This also means that

$$d.\mathcal{L}(RW(d)) = RW(d) \tag{5.1}$$

$$d.\mathcal{L}(\mathcal{T}(d)) = \mathcal{T}(d) \tag{5.2}$$

Lemma 5.1

$$\mathcal{A}_L(P) = P.d.\mathcal{L}(RW(P)) \cup \bigcup_{\forall C_j \in P.d.\mathcal{L}(\mathcal{T}(P))} \mathcal{A}_L(C_j).$$

Proof: We use D^* to represent the set of domains consisting of D and all its descendants.

This equation simply expands Axiom 5.1 to account for limits imposed by entry points. We therefore show its correctness by showing that the only two cases in which $\mathcal{A}_L(d)$ differs from $\mathcal{A}(d)$, are correct.

First, according to Axiom 5.2, if there exists more than one entry point to $P.d$, or the

entry point is unverified, then we substitute

$$P.d.\mathcal{L}(RW(P.d)) = RW(P.d) \quad (5.3)$$

$$P.d.\mathcal{L}(\mathcal{T}(P.d)) = \mathcal{T}(P.d) \quad (5.4)$$

into the above equation, which then collapses back to that of Axiom 5.1.

Now let us assume that for some domain $Q \in D^*$, there is only one, verified, entry point, and it restricts $P.d$'s access such that $\text{rwx} \rightarrow \text{root_t} \notin RW(Q)$. There are two possibilities. If no other domain in D^* may receive $\text{rwx} \rightarrow \text{root_t}$ access, then clearly P cannot ever receive this access right.

Alternatively, some domain $R.d \in D^*$ does have $\text{rwx} \rightarrow \text{root_t}$ access. Then this access will be contained in $R.d.\mathcal{L}(RW(R.d))$, so that $\mathcal{A}_L(P)$, into which $R.d.\mathcal{L}(RW(R.d))$ is unioned, will also contain $\text{rwx} \rightarrow \text{root_t}$.

The argument for the validity of $\mathcal{L}(\mathcal{T}(P.d))$ takes the exact same form as that for $\mathcal{L}(RW(P.d))$. ■

It seems likely that, in most cases, the entry point will simply act as a barrier. Certainly this would be the safest behavior, least likely to be subject to programming errors.

Equations 5.5 and 5.6 are an example of \mathcal{L} acting as a barrier.

$$\mathcal{L}(RW(P)) = \begin{cases} \phi & \text{if } P.pwd \neq \text{"god"} \\ RW(P) & \text{otherwise} \end{cases} \quad (5.5)$$

$$\mathcal{L}(\mathcal{T}(P)) = \begin{cases} \phi & \text{if } P.pwd \neq \text{"god"} \\ \mathcal{T}(P) & \text{otherwise} \end{cases} \quad (5.6)$$

We see that careful analysis of entry points can provide more precise values of $RW(d)$ and $\mathcal{T}(d)$. We will take advantage of this feature in Section 6.2 in order to decrease the number of false positives when searching for dangerous transitions.

Chapter 6

Policy Administration Tools

The configuration language read by the LSM DTE module is defined in Section 3.5. It is modeled after TIS' DTEL DTE policy language [2], which is very intuitive, well organized, and concise, such that one can reasonably understand a policy by reading its definition. However, the policy must specify a large number of relations between various domains, and between domains and types. Therefore, no matter how well the policy definition language is thought out, there will be certain problems with dealing with the policy files directly which cannot be surpassed.

- Repetitive typing

For any policy which segments the file system into a reasonably large number of types, access will likely need to be specified from most domains to most types. This involves retyping each typename up to $|D|$ times, along with as many somewhat cryptic yet repetitive access types, such as `rxld`. Furthermore, each typename is listed once in the type enumeration line, and must be bound to real objects at least once among the type assignment lines. Clearly, the probability of making a typographical error is not insignificant. In the best case, such an error will result in an unbootable system. In the worst case, it will result in a system which runs fine, but under an erroneous and

dangerous security policy.

- Dense text

The policy language is very concise. This allows a quick scan of a portion of the policy file to give a good feel, for example, for the structure of a domain. However, providing a large amount of information in a small amount of space can serve to mask a syntactic or semantic error. For instance, a missing `\` to continue to the next line might be easily missed, or a missing or extraneous `d` among the `rxl` type access can be hard to spot. An alternative would be to make the policy language very verbose.

For instance, a domain definition could be

```
domain login_d begin
entry types begin
login_et
entry types end
type access begin
read,write,exec to login_et
...
type access end
domain login_d end
```

However this could serve just as well to mask errors. Most domains definitions would likely be split among several pages, preventing related keywords from being seen together. This makes it harder to match `begin` and `end` statements. Perhaps more importantly, the expansion of domain definitions would make it much harder to un-

derstand a domain. This is both because the number of keywords begin to overwhelm the number of meaningful identifiers, (access types, typenames, and domain names), and because the process of looking through several pages to find information to define a single domain interferes with what is known as interface zen [8]. In other words, by forcing a policy administrator to look through several pages, the train of thought which was working toward understanding the policy is being interrupted.

- Visual presentation

Clearly a text file can provide exactly one visual presentation. By setting up macros in a text editor, it is possible, for example, to automatically follow a domain transition definition to the definition of the destination domain. However, several more ideal presentations come quickly to mind.

The DTE policy defines new classes of subjects and objects, namely types and domains, and defines relations between these, as well as between these and existing subjects and objects. For instance, between domains are connections indicating allowed auto and exec domain transitions, as well as connections indicating permitted intra-domain signals. There exist also connections from domains to types indicating which types may be executed to enter a domain, and more domain to type connections indicating which types a domain may read, modify or execute. One must consider a combination of these connections in order to analyze how domains may affect each other.

Understanding a policy requires understanding all of these connections. Different views of a policy, therefore, may show different sets of connections, from different

viewpoints.

An intuitive way to think about a policy is as a directed, labeled graph. The nodes represent domains, types, and files. Edges may represent four types of entities. Edges from domains to other domains are labeled “auto,” “exec,” or with a set of signal numbers, and represent either a permitted domain transition or signal rights. The domain transition edge labels might optionally refer to the entry types which may be used to effect the transition. Edges from domains to types may be labeled “e” for entry type, as well as any subset of “rwxlcda” to describe domain to type access as described in Section 3.5. Edges from types to files represent “-e,” “-u,” or “-r” type assignment rules. Finally, Edges may exist between files, representing the file system layout. An edge from file `file1` to `file2` tells us that `file2` is a child of `file1`. The last type of edge is only partially a result of the policy — as a result of the “pretend” mount rules described in Section 3.2.2 — but is certainly a part of a complete policy representation.

We define F as the set of files, T as the set of types, and \mathcal{D} as the set of domains. The graph $\mathcal{G} = (V, E)$, where $V = (\mathcal{D} \cup T \cup F)$, and E contains the edges we described.

Formally:

$$\begin{aligned}
 E = & (\forall d_1, d_2 \in \mathcal{D}, l \in (\text{auto}, \text{exec}, (0 \dots 31)^*)) : (d_1, d_2, l) \\
 & \cup (\forall d \in \mathcal{D}, t \in T, z \in \{\text{r}, \text{w}, \text{x}, \text{l}, \text{c}, \text{d}, \text{a}, \text{e}\}) : (d, t, z^*) \\
 & \cup (\forall f_1, f_2 \in F) : (f_1, f_2)
 \end{aligned}$$

Displaying a DTE policy in this format would be far more useful to policy analysis than the policy file itself. However, the full graph would be overwhelming. The policy administrator must be allowed to look at subsets of this graph which emphasize particular connections or sets of connections. One of the tools which we will present does just this, presenting subsets of this graph which we have ourselves found useful in policy analysis.

- Error patterns

As discussed above, a DTE policy defines many connections between subjects and objects. In some cases, connections of two or more types should not exist simultaneously. However, these connections may be defined in different sections of the policy, making them hard to spot. Or, the sheer number of these connections may make it impractical to spot an inappropriate pair by eye. Consider that, if we have ten connections of some type, but one pair of connections is not appropriate, we must consider $\Sigma_{i=1}^9 i = 45$ pairs. In Section 7.4, we will present a very minimalistic policy, providing only enough detail to support an intelligent login daemon. Even in this policy, shown in Appendix A.1.1, the number of domain to type access rules is 86, each of which specifies between one and seven type accesses taken from the set $\{r, w, x, l, c, d, a\}$ as described in Section 3.5.

In building DTE policies, we have found several error patterns resulting from such interactions of connections. While they are hard to spot by eye, they are simple to find automatically.

1. Conquering

We say that a domain D_1 can conquer another domain D_2 provided that

- (a) D_1 may transition to D_2
- (b) e is an entry point to D_2
- (c) D_1 may write or replace e

Clearly, in this case, we can say of the privileges of D_1

$$\mathcal{T}(D_1) = \mathcal{T}(D_1) \cup \mathcal{T}(D_2)$$

$$RW(D_1) = RW(D_1) \cup RW(D_2)$$

since any actions which D_2 is allowed to perform, D_1 could also perform, by writing the instructions into e , and requesting a domain transition to D_2 upon execution of e .

2. Trojan

Entry points are a domain's only means of protection from untrusted code. For some domains, protection from untrusted code is moot, since they are meant to run shells and user-compiled or user-written code. The `user_d` domain presented in Section 7.4 is an example. However, many domains will be designed to temporarily expand a user or daemon's access rights while performing a specific, restricted task. In such cases, the domain's entry points must be designed such that the domain's privileges cannot be used for any unintended purposes. An attack wherein a system is tricked into executing untrusted code is called a Trojan horse attack.

For domains whose entry points are untrusted, we may wish to check for any

pathnames to which the domain has execute access, and which the domain may itself replace. Since replacing the pathname can mean overwriting the file itself, or writing to any of its parent directories, this is clearly a check best performed automatically.

3. Insufficient entry type access

A domain which cannot be entered is a useless domain. A domain cannot be entered if it cannot execute its own entry points, or if it cannot descend the file system tree down to the entry points. This will cause denial of service to either user domains or system services. Furthermore, we have pointed out the danger of vulnerability to Trojan attacks. The usefulness of Lemma 5.1 depends entirely upon carefully considered entry types, which must not be vulnerable to attack. Any domain with insufficient entry type access likely has not been sufficiently analyzed. For these reasons, automated checks for sufficient entry type access for all domains is desirable.

Our policy analysis tool will detect the presence of these conditions. In the following sections we present `DTEedit` and `DTEview`, which, together address each of the above concerns. A third tool, `DTEbuild`, will be presented in the next chapter.

6.1 `DTEedit`

`DTEedit` is a Gtk-enhanced graphical user interface for creating and editing DTE policies. It is mainly intended to address the excessive typing and dense text problems associated with editing policies in text format. Typing is dramatically reduced by asking the user to enter

type and domain names exactly once, to inform DTEedit of their existence. Thereafter, specification of access to or from any domain or type is by list selection, as is selection of type of access (`rwxclda` for type access, and `auto` or `exec` for domain access). When entering type assignment rules, pathnames can be selected by browsing the file system using a file selection dialog, or by typing the pathname. Figure 6.1 demonstrates specification of `exec` domain transition access from `login_d` to `root_d`.

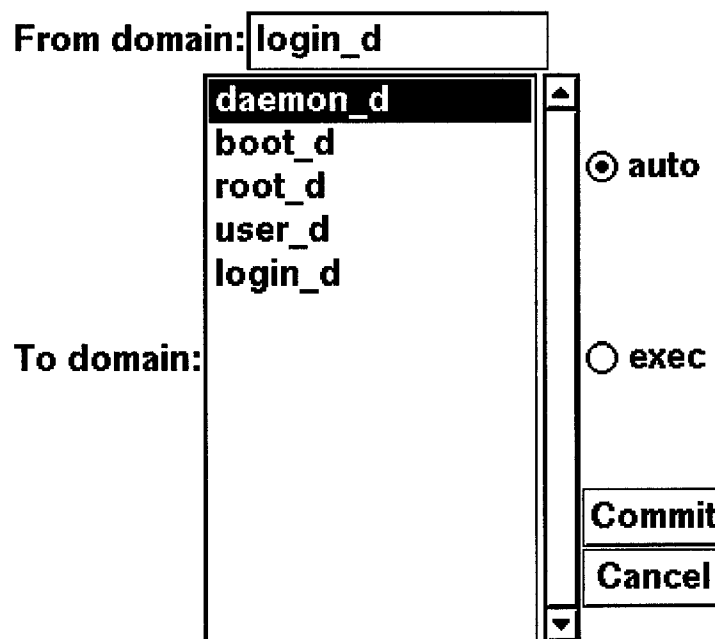


Figure 6.1: Adding new domain specification rule in DTEedit.

The problem of dense text is solved simply by presenting only a small piece of a policy

at once. In Figure 6.2, for instance, we see only the domain transitions permitted out of domain `login_d`. Furthermore, syntactical symbols are not necessary, so the `->` separating access specification from domain name, and parentheses, are not necessary, as they are replaced by the structure of the domain specification table.

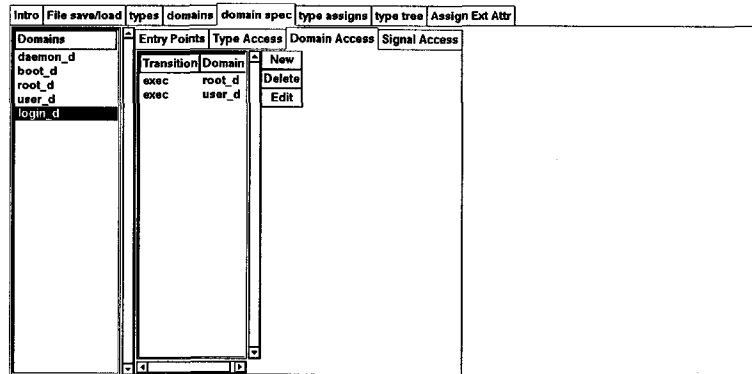


Figure 6.2: Viewing a domain specification in DTEedit.

For the most part, DTEedit is not helpful in presenting the policy in new formats. A domain specification is still entered and viewed as a set of entry types, a set of domain transitions, a set of type accesses, and a set of signal accesses. One exception is for viewing type assignments. A long list of type assignment rules is simply not useful in understanding the resulting type assignment tree. Therefore, DTEedit provides a file manager which walks the host machine's file system, and displays the results of applying the currently entered type assignment rules to the file system.

DTEedit is meant to provide a safer alternative to manually typing policies. Many people far prefer typing to excessive use of the mouse. Nevertheless, the use of DTEedit is strongly encouraged for the sake of reduced typographical, syntactic, and semantic errors, as well as a better understanding of the policy during construction.

6.2 DTEview

DTEedit addresses our first two concerns with usage of text-file policies for editing and analyzing policies. The latter two concerns are addressed by DTEview. DTEview is a Perl/TK tool which begins by detecting any error patterns discussed on page 91, and warning the administrator if they are present. Two of the error patterns, namely insufficient entry point access and conquering, are always warned against. The third, the ability by a domain to overwrite and execute a type, is acceptable in very many cases, so that warning of all instances would provide enough false positives that a policy administrator would likely ignore all such warnings. Therefore, the administrator may tell DTEview of any domains about which he is concerned. This is done by adding them to an array called `@paranoid_wx` in the file `restrictions.pl`. Any domains listed in this array will be checked to ensure that there is no type which they can both overwrite and execute. Figure 6.3 shows such a DTEview warning.

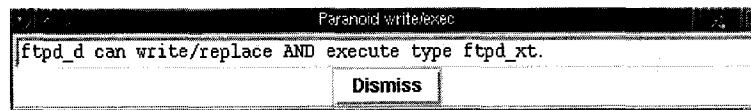


Figure 6.3: An error pattern popup warning in DTEview.

DTEview goes on to provide three ways of viewing the policy. The first presents a tool acting like a file-manager, but showing the DTE type information for files and directories. In addition, this tool displays the pathname which DTE would actually use for typename resolution. Recall that due to mounting activity and mount restrictions, this may be different from the given pathname. Figure 6.4 shows the file-manager view in DTEview.

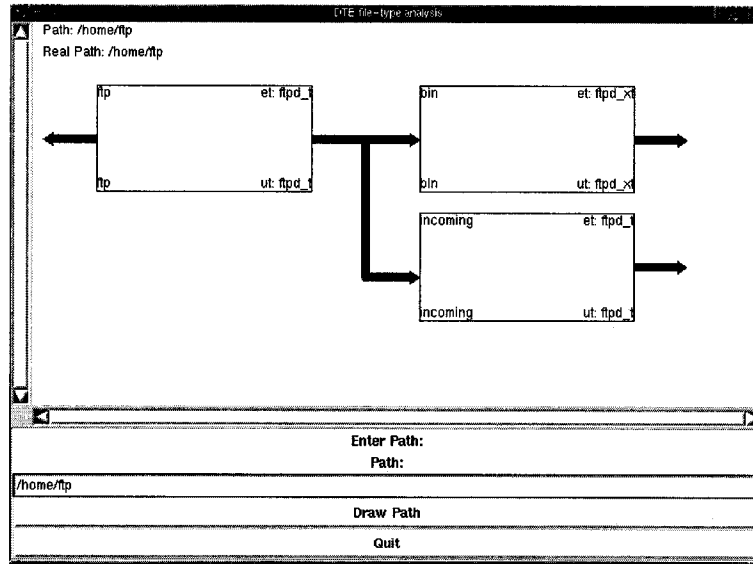


Figure 6.4: The file-manager tool in DTEview.

The second view presented by DTEview is a domain transition analysis. It begins with the first process (`/sbin/init`) running under the default domain. It also displays all domain transitions permitted from this domain, and all entry types through which the transitions may occur. The resulting type/domain pairs are shown as children of the first process in a tree. From here, at any level, one domain may be chosen to be expanded, showing either the permitted domain transitions, or the permitted type accesses. Clearly, we are presenting a restricted view of the full policy graph \mathcal{G} . Another way this could be presented would be to keep the graph structure. In fact, another tool which was created presented a 3-D fly-through universe view of a policy. However, while perhaps more amusing than the domain transition tree view of DTEview, it was no more informative, and quite a bit more confusing. Again, we wish to present simplified, clearer views of the policy. The 3-D universe policy view would be much more useful as an initial policy view. After gaining an

initial understanding of the policy in this way, the user could select a focused view, like the domain transition tree, to concentrate on a particular feature or problem.

Since the number of type accesses is typically very large, when type accesses are displayed, DTEview offers a bit of help. First, type accesses are sorted by typename. Second, any write or create accesses are flagged in red. This highlights the more dangerous accesses which an administrator is likely interested in. Third, middle-clicking on a type will bring up a list of all paths to which this type is assigned. Finally, DTEview is able to present a filtered view of the type accesses. For instance, the administrator may request only those type access containing `rwa`, that is, read, write, and append, access. Figure 6.5 shows the domain analysis view, with type accesses out of `daemon_d` filtered to show only types to which `daemon_d` has full (`rxwlc`) access.

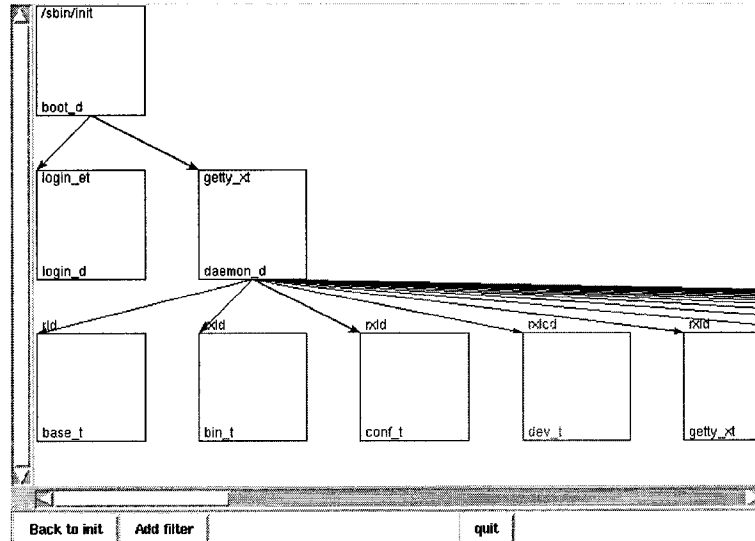


Figure 6.5: Domain transition analysis in DTEview

The third view is based upon reachability queries. There are two types of queries. The

first searches through \mathcal{G}_1 .

$$\mathcal{G}_1 = (\mathcal{D}, E_1)$$

$$E_1 = (\forall d_1, d_2 \in \mathcal{D}, l \in (\text{auto}, \text{exec})) : (d_1, d_2, l)$$

For the query $Q(d_1 \in \mathcal{D}, d_2 \in \mathcal{D}, n \in I)$, DTEview returns all paths from d_1 to d_2 containing fewer than n edges. This corresponds to all the ways in which a process under domain d_1 can make at most n domain transitions to end up in domain d_2 . The second type of query searches the larger graph \mathcal{G}_2 .

$$\mathcal{G}_2 = (\mathcal{D} \cup T, E_2)$$

$$E_2 = (\forall d \in \mathcal{D}, t \in T, a_t \in \{\text{r}, \text{w}, \text{x}, \text{l}, \text{c}, \text{d}, \text{a}\}) : (d, t, a_t^*)$$

A query $Q(d \in \mathcal{D}, t \in T, n \in I, a_t \in \{\text{r}, \text{w}, \text{x}, \text{l}, \text{c}, \text{d}, \text{a}\})$ returns paths originating at d and terminating at any domain d_2 representing a domain which has type access a_t to type t .

DTEview finds all paths satisfying a query and displays the sequence of domain transitions for the first path. An example is shown in Figure 6.6. The user can step through all the paths, and may, at any point, click on the displayed path to bring it up in the domain transition analysis view for further analysis. While working back in the domain transition analysis view, the user may right-click on a domain to select it as the source domain, or on a type to select it as the target type, in the reachability query. In this way we attempt to offer simplified views of the policies, while still allowing quick switching from one view to a place of interest in the other.

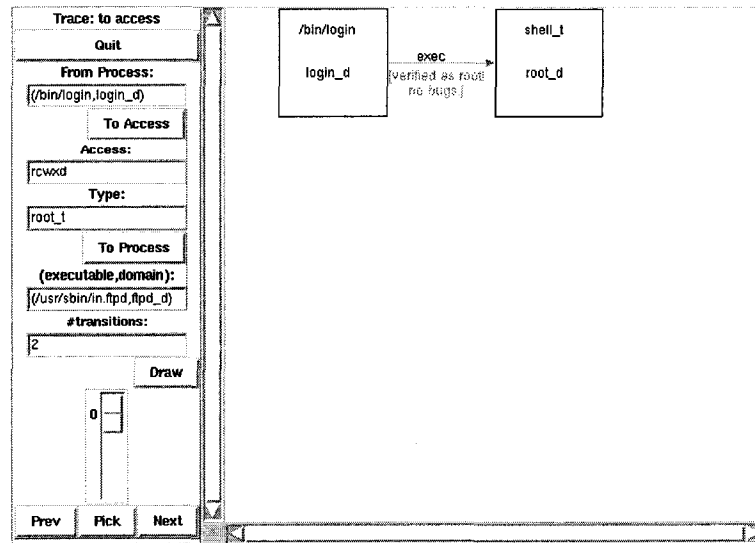


Figure 6.6: A reachability query in DTEview.

Normal queries take into account only the DTE policy file. Queries therefore calculate access rights of domains according to Axiom 5.1. Lemma 5.1 limits the access rights of domains based upon the properties of entry points to other domains. DTEview assertions provide the same power to queries.

An entry point is a file whose execution may be used to trigger a domain transition. While the DTE policy specifies whether a domain transition from d_1 to d_2 is permitted, the entry point(s) to d_2 may consider additional system parameters. Based upon these, it may choose to refuse entry, allow restricted entry, or allow full access.¹ A DTEview assertion is intended to describe an entry point's behavior. The assertion takes the form:

`{d:ind:outd", <ACTION>}`

¹Note that it may only restrict access using UNIX features. The set of DTE access rights for d_2 is not malleable, but the entry point to d_2 can simply refuse to execute system calls leading to violation of some set of rights.

where *ind* is the source domain, *outd* is the destination domain, and *d* indicates that this assertion pertains to domain transitions. “<ACTION>” may be one of the following:

<ACTION> ::= IGNORE

<ACTION> ::= IGNORE_SAY <STRING>

<ACTION> ::= SAY <STRING>

<ACTION> ::= REJECT

Some transitions likely should be entirely forbidden. For example, since we do not trust the binary `/usr/sbin/in.ftpd`, we do not wish it to enter the `root.d` domain under any circumstances. The REJECT action is intended for such a situation. REJECT strings are checked and warned against when DTEview starts up. These will become far more useful in the next few chapters, when we begin to build policies from components which are joined using generic access rules.

On the other hand, if we search for dangerous transitions, we may not wish to be distracted by transitions which we know to be safe. For instance, the binary `/sbin/login` may be a modified version of `/bin/login`, whose code has been verified not to allow root logins unless the login occurs on console, in a locked room under heavy guard. In this case, we may wish for transitions from `/sbin/login` into the `root.d` domain to be ignored.

Clearly, an ignore action `{d:ind:outd:IGNORE}` should be used only when it is known that $outd \notin ind.\mathcal{L}(\mathcal{T}(ind))$. An action which we believe more useful is IGNORE_SAY. When this action is tied to a transition from domain *ind* to domain *outd*, any paths containing this transition are still shown. However, STRING will be printed above the arrow representing the transition. STRING is meant to be a brief description of $ind.\mathcal{L}$.

For instance, if \mathcal{L} has been verified to allow logins only from a physically secure local console, then STRING might be

`Verified denied unless on local console`

The policy administrator may choose whether the remaining threat is relevant to his current query. If not, he may ignore it. Else, he may study it. While an SAY action labels the associated transition with the specified string, an IGNORE_SAY action also changes the color of the edge to indicate that this transition may most likely be ignored, subject to the condition specified in the string.

Before administrators use this mechanism to analyze security policies, it must be shown that proper use of these assertion labels will not cause a query to return incomplete results. We begin with a rather obvious axiom:

Axiom 6.1 *Let \mathcal{G} be a domain transition graph representing some policy. Let d_1 and d_2 be any two distinct domains in the policy, such that d_1 may not transition to d_2 . That is, no edge from d_1 to d_2 exists in \mathcal{G} . Let \mathcal{X} be the set of assertions relating to \mathcal{G} , and let $\xi \in \mathcal{X}$ be an assertion relating to a transition from d_1 to d_2 . Then any query made under \mathcal{X} will produce the same results as the same query made under $\mathcal{X} - \xi$.*

In other words, an assertion on a domain transition which the policy does not allow, will not affect the outcome of any queries.

In the following theorem, we continue to define the graph of all domain transitions as \mathcal{G} . The graph containing the domain transitions as allowed by $\bigcup_{d \in \mathcal{D}} d.\mathcal{L}$, that is, the graph containing all domain transitions which are allowed by the entry points, is \mathcal{G}' . \mathcal{X} is the set of assertion labels read by DTEview, and ξ is an individual assertion label in \mathcal{X} .

Theorem 6.1 *A query for transition paths from domain d_1 to domain d_2 containing fewer than n domain transitions, with assertion labels in \mathcal{X} correctly reflecting \mathcal{L} for all entry points, will return all possible paths from d_1 to d_2 in \mathcal{G}' .*

Proof:

We will prove this by induction over the number m of assertion labels. Our base case:

We take it for granted that, in the absence of any assertions, DTEview will correctly return all paths from d_1 to d_2 containing fewer than n transitions. This is a matter of correctly coding a simple graph algorithm.

Inductive Step: Assume that for a query under $m - 1$ assertions, DTEview returns all paths in \mathcal{G}' . Then for a query under m assertions, DTEview returns all paths from d_1 to d_2 in \mathcal{G}' .

As defined above, an assertion can be of the following types:

1. REJECT:

A rejection is only checked at the start of DTEview. It does not affect the search for paths to satisfy a query.

2. SAY:

This type of assertion only returns extra information, displayed above certain domain transitions. It cannot prevent a path from being returned in response to a query.

3. IGNORE:

If an ignore for a transition from domain d_a to d_b is correctly applied, this means the following:

- (a) d_a has only one entry point

- (b) The entry point to d_a cannot be overwritten by any domain.
- (c) The code of the entry point to d_a must be verified such that, while the transition from d_a to d_b may in fact be permitted by the policy,

$$d_b \notin d_a.\mathcal{L}(\mathcal{T}(d_a)). \quad (6.1)$$

Another way of saying this, is that the edge from d_a to d_b is not in \mathcal{G}' , although it may exist in \mathcal{G} .

Since \mathcal{G}' does not contain an edge from d_a to d_b , then, by Axiom 6.1, any query made under $\mathcal{X} - \xi$ will return the same results as the same query made under \mathcal{X} . Since $|\mathcal{X} - \xi| = m - 1$, we know, by induction, that DTEview will return this set of paths correctly.

4. IGNORE_SAY:

This case could be treated two ways. First, since paths including a transition tied to an IGNORE_SAY assertion are not excluded from query results, as paths which include a transition tied to an IGNORE assertion are, we could treat IGNORE_SAY as a SAY assertion, which is trivially shown to be safe. However, we would like to show that the administrator can take the IGNORE_SAY action at its word, in effect becoming a part of DTEview's behavior, and trust the results.

The proof of safety for IGNORE_SAY is much like that for IGNORE. The difference is that, at Equation 6.1, we must take the process and system state into account.

Equation 6.1 becomes:

$$d_b \notin P.d.\mathcal{L}(\mathcal{T}(P.d)). \quad (6.2)$$

There are now two cases. If the state of P is such that $P.d.\mathcal{L}$ allows the transition, then correct application of the assertion label means that the administrator does not ignore the transition. The assertion is therefore trivially safe. If the state of P is such that $P.d.\mathcal{L}$ does not allow the transition, then Equation 6.2 collapses to Equation 6.1. In this case, we can revert to the proof of safety for IGNORE labels.

To make this more concrete, a quick example. In the case of an assertion

```
{d:da:db:IGNORE_SAY "Verified denied unless on local console"}
```

the administrator must decide whether logins on local console fall into the threat which he is currently assessing. If so, then he will study paths including transitions from da to db . If not, then he ignores these paths.

■

In Lemma 5.1, we see that the other function of \mathcal{L} is to limit $RW(d)$. Therefore, we would of course like to use the following type of label as well:

```
{t:ind:access,outr", <ACTION>}
```

However, this is dangerous. Recall Axiom 5.1, which states:

$$\mathcal{A}_L(P) = P.d.\mathcal{L}(RW(P)) \cup \bigcup_{\forall C_j \in P.d.\mathcal{L}(\mathcal{T}(P))} \mathcal{A}_L(C_j).$$

For instance, assume that we created the label $\{t:ind:rw->t1:IGNORE\}$. Now, even though the label is presumably correct, such that `ind` is in fact denied `rw->t1`, this only limits $P.d.\mathcal{L}(RW(P))$. In order to obtain the full $\mathcal{A}_L(P)$, we must union this with access rights for all domains which may be reached through $P.d$. The `IGNORE`, `IGNORE.SAY`, and `SAY` labels are therefore not safe for type access assertions. However, we do support the `REJECT` action. This is, again, a simple and useful way of ensuring some basic properties about a complex policy which could be the result of automated composition of several pieces.

This chapter presented tools designed to solve some of the obvious problems encountered while editing and viewing policies. These tools have not gone beyond the traditional concepts of DTE. The next two chapters present methods for analyzing and creating policies, culminating in a novel method for policy composition.

Chapter 7

Analysis of DTE Policies

In chapter 6, we presented DTEview as a tool designed to aid the analysis of DTE policies.

We now present some further methods for analyzing policies.

In a paper [23] about the safe run-time extension of DTE policies, TIS presented the idea of analyzing DTE policies using simpler, well-understood access control policies. Since the policies expressible by DTE policies are a superset of those expressible by many traditional access control policies, it is possible to search for properties of those traditional policies which are exhibited in a particular DTE policy. TIS used this concept by asserting that any such properties exhibited in the DTE policy must not be violated by modifications to the policy. We feel this concept is also useful for the analysis of a static DTE policy.

We will begin in the next section by showing how searching for properties of the Bell-La Padula access control policy in a DTE policy can yield useful information. Section 7.2 will discuss the limitations of this first attempt, and section 7.3 proposes a far more powerful, yet still very simple, extension of the Bell-La Padula policy. Here we prove that this modified BLP policy is capable of expressing the Clark-Wilson and pipeline policies. Finally, section 7.4 applies this idea to several DTE policies.

7.1 Using the BLP *-Property

The Bell La Padula access control system, discussed in section 2.2.1, maintains two access rules. The simple security rule relates the security level of objects to those of subjects. We only use BLP to relate objects, and therefore ignore this rule. The second rule is to maintain the *-property, which dictates that if a subject may observe object O_1 , and also may modify object O_2 , then the security level of O_1 is less than or equal to that of O_2 . If this is not the case, then the subject is able to leak information from a higher security level to a lower security level. A second subject, permitted to observe O_2 but not O_1 , can then access information in O_1 with the aid of the first subject.

We will use the *-property to introduce a partial relation on types in a DTE policy. Figure 7.1 shows the algorithm used to calculate the BLP less than relation from a policy file.

```
leqlist = {};
for each domain D do
    for each type T1 which D can read do
        for each type T2 which D can write do
            add "T1:T2" to leqlist
        done
    done
done
for each string in leqlist do
    print string
done
```

Figure 7.1: Algorithm to calculate BLP $<$ relation

We define observe access as r , or a simple file read. That is, we do not consider directory read or directory descend to be observe accesses for this analysis. However, we define modify access as file write (w), file append (a), and directory create (c). In addition, modify access

to any parent directory of a file f constitutes modify access to f itself, since f can be replaced with an entirely new copy.

We would like to consider what information may be gleaned from the BLP relation applied to the types of a DTE policy. The three strongest results which we may find for a type in the context of a given policy are that it is unrelated to all other types, that it is strictly less than all other types to which it is related, or that it is strictly greater than all other types to which it is related. Each of these cases can be shown to convey important information regarding the security or integrity properties of the type.

1. Type T_1 is unrelated to all other types.

In this case there exists no domain permitted to access both T_1 and any other type. Therefore no one is able to corrupt the data in T_1 using data from any other type,¹ and no one may leak the data from T_1 to any other type.

The integrity claim may not prevent subjects from erasing data from T_1 , or replacing the data with all 1's. However, Trojan horse attacks, as a particular example, are nearly impossible, as the Trojan horse cannot be read from any other types. The attack is possible only if the Trojan horse is hard-coded into the subject's source code. Verification of entry points, and assurance that entry points cannot be replaced, will prevent this final Trojan horse attack.

2. Type T_1 is strictly less than all other types to which it is related.

There exist domains which may read type T_1 , as well as modify other types T_i . There-

¹Note that, in Unix, all devices are files, and hence even random data (from `/dev/random`) must come from files of some type.

fore data from T_1 can be leaked to T_i . However, there exists no domain which may write T_1 and also read another type. Therefore, data may not be moved from any other types to T_1 . We consider T_1 to be of high integrity.

3. Type T_1 is strictly greater than all other types to which it is related.

There exist no domains which may read T_1 and write other types. Information from T_1 cannot be leaked to any other types. This is a strong secrecy claim. However, there exist domains which may modify T_1 while reading other types. The integrity of T_1 is therefore shown to be suspect.

By imposing a BLP relation onto a DTE policy, and searching the relation for the above three conditions, we hope to provide some automated analysis of DTE policies.

7.2 Limitations of BLP

The BLP policy is a simple one. This is useful in that it allows us to introduce a simple relation on types. However, a consequence of its simplicity is a lack of expressiveness. BLP works well if we can keep security domains completely segregated. By this we mean that no domains are provided access to the same types. Consider a top secret type T_S , to which only one domain D_S has read access, and no domains have write access. This domain may need to warn other domains of certain conditions, for instance corrupted data under T_S . This requires write access to some type T_W , to which other domains have read access. By BLP,

$$T_S \leq T_W.$$

If there exist any domain which has read access to T_W , then, by transitivity, $T_S \leq T_i$ for all types T_i to which these other domains have write access. Consequently, information can be leaked from T_S to each such T_i . In order to prevent this, we must prevent all other domains from having write access to any $T_j : T_S \leq T_j$. In other words, we are segmenting D_S such that all types to which it can write are unreadable by all other domains. A graph of the BLP relation for the types of a DTE policy which segments a domain D_S in such a way, is shown in figure 7.2. In this particular case, the types TS, TX and TY may not be accessed by any domains other than D_S , which itself may not access any types other than these three.² The dotted arrow from `user_t` to TS indicates that D_S may have modify access to TS and observe access to `user_t`. In that case, the integrity of the `secret_group` group is affected, but not its secrecy.

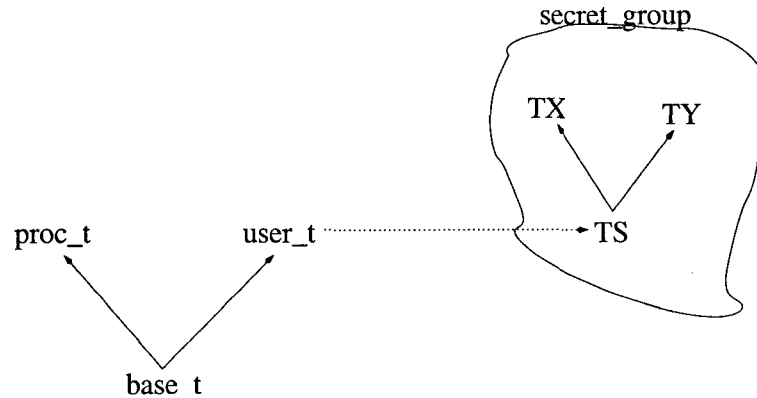


Figure 7.2: BLP for policy excerpt with disjoint type group.

Let us assume there is a type, T_S , whose secrecy properties we would like to analyze. It is possible to create a domain which may read T_S , but may not write any of the common

²This is not strictly necessary. In fact, other domains are restricted only in that if they have observe access to T_S , T_X or T_Y , they may not have modify access to other types, and vice versa.

types, such as files under `/proc` and `/dev`. However, such access may be required. As Lemma 5.1 shows, the domain's access rights may also be limited by a single effective, verified entry point. In this case, we can label this domain a trusted domain. This means that the domain exists outside the BLP policy. Due to BLP's lack of expressiveness, this is the only way to express this concept. However, by completely exempting the domain from the BLP policy, the domain becomes far too powerful.

Consider the assured pipeline (see Section 2.2.5) shown in Figure 7.3. If `syslog_d` is allowed to write `dev_t`, then, since it may read `in_log_t`, $\text{in_log_t} \leq \text{dev_t}$, instantly lowering it to the same security level as most types on the system.

If we define `syslog_d` as a trusted domain, `in_log_t` becomes strictly greater than all other types, since all other domains may write to it, while reading common types. However, we now lose much useful information regarding `syslog_d`. For instance, we may have another assured pipeline, whose information is transformed by another domain. By defining both domains as trusted, we can no longer identify cross-talk between these pipelines. Furthermore, we cannot express the concept that `syslog_d` should not be able to write its entry points. The complete resistance of entry points of `syslog_d` to subversion will surely be a condition of any trust we place in `syslog_d`.

7.3 Modified BLP

We have shown the need to increase the expressiveness of the BLP relation, as well as the insufficiency of trusted users as a means of addressing this deficiency. We now present a more powerful way of expressing concepts such as assured pipelines in BLP. Our modified

BLP, or MBLP, will be made more expressive than BLP through the addition of two simple concepts: secrecy with exception, and integrity with exception. These concepts will be implemented in analysis by a set of two types of statements, which are to be read alongside a DTE policy. The two statement types are:

```
secret <type1> except from <domain_1_1> [,<domain_1_n>]
```

and

```
protect <type2> except from <domain_2_1> [,<domain_2_n>]
```

These provide four hints for analysis. The first is our explicitly intended purpose. Namely, when building our list of types which domain `domain_1_1` may read, we do not add `type1` to the list. This means that, if `domain_1_1` has write access to some type `type_1w`, then we will ignore the BLP `*`-property, and we will not use this fact to define `type1 ≤ type_1w`. Of course, it does not prevent us coming to this same conclusion through some other domain's type accesses.

Second, they indicate that a separate check should be made to ensure their correctness. The DTE policy should deny all domains (except `domain_1_1` through `domain_1_n`) read access to `type1`. Likewise, all domains (except `domain_2_1` through `domain_2_n`) should be denied modify access to `type2`. If this is not the case, a warning flag should be raised.

Third, as a consequence of the previous two hints, while building the read list for any domain, we can ignore `type1` altogether. If we find that a domain has observe access to `type1`, then there are only two possibilities. Either the domain is listed as an exception to `type1`'s secrecy, in which case we are instructed to ignore it. Otherwise, the domain is not listed as an exception. In this case, the domain is in fact not allowed to read `type1`,

a condition we have already enforced under the second hint. Therefore, we know that any case wherein a domain is allowed to read `type1` is irrelevant to the $\text{MBLP} \leq$ relation.

Finally, since the domains listed as exceptions to security and integrity declarations are being provided an extra measure of trust, we wish to ensure that they are worthy of such trust. We therefore check that the entry points to all such domains, in addition to all trusted domains, are themselves protected, and inform the policy administrator that the code of the entry points must be verified.

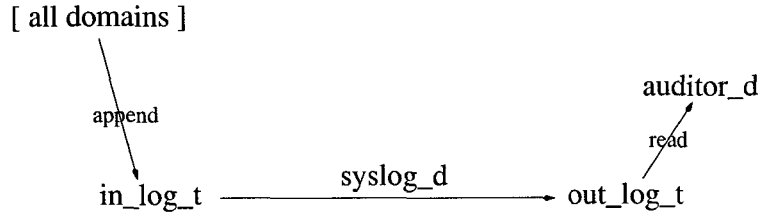


Figure 7.3: Sample Assured Pipeline

A precise definition of the $\text{MBLP} \leq$ relation follows.

$$\forall x \in \mathcal{D} \text{ and } a, b \in T : mr(x, a) \wedge mw(x, b) \Rightarrow L(a) \leq L(b) \quad (7.1)$$

where

$$mr(x, a) = r(x, a) \wedge \overline{re(x, a)} \wedge \overline{id(x)} \wedge \overline{it(a)} \quad (7.2)$$

$$mw(x, a) = w(x, a) \wedge \overline{we(x, a)} \wedge \overline{id(x)} \wedge \overline{it(a)} \quad (7.3)$$

Here $re(x, a)$ means that domain x is on the read exception list for type a . Likewise, $rw(x, a)$ means domain x is on the write exception list for type a , while $id(x)$ and $it(a)$ mean that domain x and type a , respectively, are trusted (i.e., to be ignored).

7.3.1 MBLP Enforces Clark-Wilson CDIs

The `secret` and `protect` keywords are in the spirit of trusted users in BLP and ring policies. However, they can also be used to implement Clark-Wilson and assured pipeline policies. We show the former here, and the latter in Section 7.3.2.

To show that Clark-Wilson policies can be implemented using these two rules, we begin by describing a policy excerpt whose purpose is to enforce a Clark-Wilson policy. We then show a set of rules which allows us to express the required properties. As described in Section 2.4, a Clark-Wilson policy specifies constrained data items (CDIs), which are data which may be modified only by certain sets of transformation procedures (TPs). We will implement this as follows. For each of the n CDIs, we will define a type `cdi_ti` \in (`cdi_t1..cdi_tn`), and a domain `cdi_di` \in (`cdi_d1..cdi_dn`). The entry points to each domain will be the TPs which are allowed to modify the CDI. The policy excerpt is shown in Figure 7.4

```
types ... cdi_t1 ... cdi_tn cdi_d1_et ... cdi_dn_et ...
domains ... cdi_d1 ... cdi_dn ...

spec_domain cdi_d1 (1 cdi_d1_et) (... w->cdi_t1) (0) (0)
...
spec_domain cdi_dn (1 cdi_dn_et) (... w->cdi_tn) (0) (0)
```

Figure 7.4: Policy excerpt defining Clark-Wilson policy.

We specify the following n MBLP rules to check for the security of Clark-Wilson CDIs:

```
protect cdi_t1 except from cdi_d1
...
protect cdi_tn except from cdi_dn
```

We must show that two properties hold. Namely, that each CDI may be modified only by the associated TPs, and that the TPs may only be executed by authorized subsets of users.

Theorem 7.1 *Each CDI may be modified only by the associated TPs.*

Proof: Only domain `cdi_dj` can modify `cdi_tj`, and it can be entered only through its entry points, which are the TPs allowed to modify `cdi_tj`. Note that MBLP is only enforcing the domain to type access, not entry points. Since the code for all entry points must be meticulously verified anyway, we consider this sufficient. ■

Theorem 7.2 *TPs may only be executed by certain subsets of users.*

Proof: This second restriction requires policy administrators to review the TP code on a case by case basis, in order to ensure that (a) only the authorized users are allowed to complete execution, and (b) authorized users cannot cause the TP to run any other (unauthorized) code. However, Clark and Wilson found no more automated way of enforcing this restriction, and acknowledged this as a weakness of their policy, as well as of, perhaps the integrity problem as a whole. In other words, this is a feature of CDI's, rather than a weakness of DTE. ■

7.3.2 MBLP Enforces Assured Pipelines

An assured pipeline, introduced in Section 2.2.5, permits the control of data flow through a system. To show that `secret` and `protect` statements can express assured pipelines, we begin with the DTE policy excerpt in Figure 7.5. The pipeline is implemented as domain

`pipe_d` with a single restrictive (and verified) entry point. It flows data from type `src_t` into type `dest_t`. We define the following two rules:

```
protect dest_t except from pipe_d
```

```
secret src_t except from pipe_d
```

Showing that this policy implements an assured pipeline requires proving the following three statements, mentioned in Section 2.2.5.

Theorem 7.3 *Data may not flow from `src_t` to `dest_t` except by passing exclusively through `pipe_d`.*

Proof: We know that $cwa \rightarrow \text{dest}_t \notin RW(d_i)$ for all domains other than `pipe_d`, whose entry point implements the pipeline’s functionality. If this were not so, an MBLP analysis would detect and warn of another domain’s write access to `dest_t`. ■

Theorem 7.4 *The pipeline’s results cannot be reversed or modified.*

Proof: The policy ensures that $cwa \rightarrow \text{dest}_t \notin RW(d_i)$ for all domains except `pipe_d`. The policy administrator must verify the entry point to `dest_t` to ensure that it cannot be used in order to rewrite results of already completed transformations. Since `pipe_d` is listed as an integrity exception, an MBLP analysis program would warn of any domains which could overwrite or replace the entry point, so that once the entry point’s code has been verified, the entry point remains trustworthy. ■

Theorem 7.5 *Subsystem is correct.*

Proof: This is proven by the implementer of the entry point to `pipe_d`, or the policy administrator, on a case by case basis. ■

Of course, while `secret` and `trusted` statements are very expressive, we still wish to allow labeling domains as fully trusted. The type accesses allowed such a domain will not affect the calculation of the BLP $<$ relation. For instance, the following line would be added to the `halt` domain:

```
trusted domain halt_d
```

Now the fact that `halt_d` is able to read and write all domains will not necessarily cause all types to be of equal security level, as it would without either this statement, or large number of `secret` and `trusted` statements. One side-effect of declaring a domain as trusted should be for the policy analysis program to declare the trusted domain's entry types as protected without exceptions, and warn the policy administrator to verify the code of all executables which are assigned these entry types.

```
types ... src_t dest_t piped_et ...
domains ... daemon_d pipe_d ...

spec_domain daemon_d (...) (...) (auto->pipe_d) (...)
spec_domain pipe_d (piped_et) (... w->dest_t r->src_t) (...)
...
assign -e piped_et /sbin/piped_executable
```

Figure 7.5: Policy excerpt implementing an assured pipeline.

The algorithm to calculate our modified BLP relation is shown in Figure 7.6.

7.4 Examples of Analysis Using MBLP

We are now ready to analyze several policies using the modified BLP relation. When discussing the partial relation induced on types by the modified BLP, we will relate it in

```

for each secret type t do
    if untrusted domain d may read t
        ensure d is an exception to t's secrecy
for each protected type t do
    if untrusted domain d may write t
        ensure d is an exception to t's protection
for each untrusted domain d do
    readlist = NULL
    writelist = NULL
    for each type access from d do
        if access==read and not secret(type)
            readlist .= type;
        if access==write|replace and not protected(type)
            writelist .= type;
    for each readel in readlist do
        for each writeel writelist do
            define "readel <= writeel"

```

Figure 7.6: Algorithm to calculate the modified BLP relation

the form of a directed graph. Nodes in this graph will be types, or sets of types. An edge from node V_1 to node V_2 indicates that $V_1 \leq V_2$. In general, we will combine types which are equal into one node, so that, for the most part, an edge from V_1 to V_2 will actually indicate $V_1 < V_2$. Where this is not the case, the cycle will be made obvious.

We begin with what we will call our base policy, shown in Appendix A.1.1. Figure 7.7 shows the associated BLP relation graph.

In the base policy, the level of most types is equal. The only types which are not equal to all others are `base_t`, `disk_t`, `getty_xt`, `login_et`, `sbin_t`, `shell_t`. We have no information to relate any of these types to each other, however all are less than the group of all other types.

Now let us analyze a more complicated policy, which implements a `passwd` domain to which users can switch to safely change passwords. This policy is shown in Appendix A.1.2.

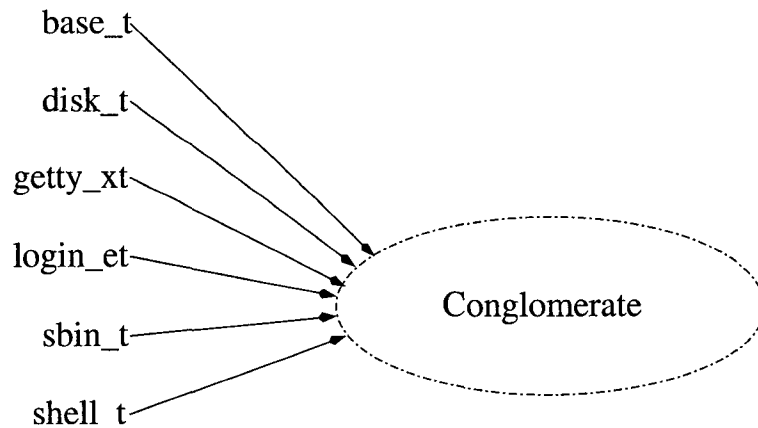


Figure 7.7: BLP less than relation graph for base policy.

If we do not specify any secrecy or protection rules, the result is the ordinary BLP relation, shown in Figure 7.8. The entry type `passwd_et` cannot be overwritten by anyone, and is therefore strictly less than or unrelated to all other types. However, `passwd_t` and `shadow_t` are both equal to most other types, since `passwd_d` may read and write both these types in addition to `log_t`, which is equal to the majority of types in the policy.

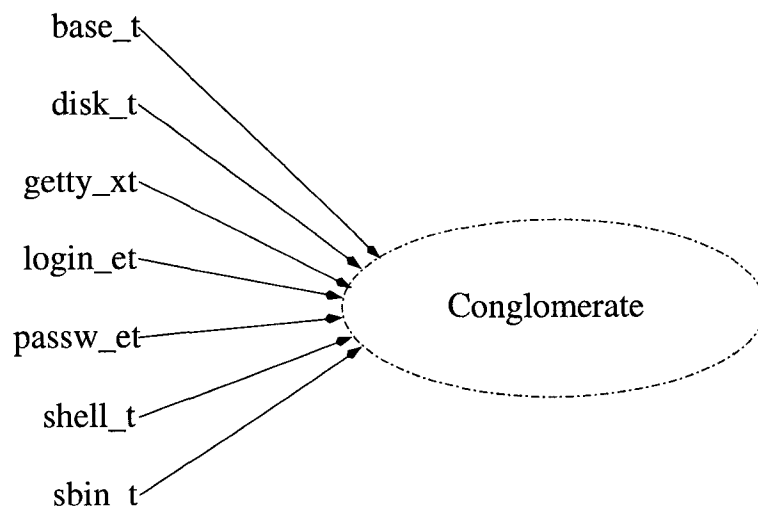


Figure 7.8: BLP less than relation for password policy.

Now let us define some protection rules to be taken into account while calculating the modified BLP. We will use the following set of rules:

```
secret shadow_t except from passw_d, login_d
```

```
protect passw_t except from passw_d
```

By making `login_d` and `passw_d` exception domains, we automatically cause our algorithm to add the following rules:

```
protect passw_et
```

```
protect login_et
```

This will provide some bit of assurance that these domains, which have been granted unusual power, will not easily be subverted.

The resulting BLP relation is shown in Figure 7.9.

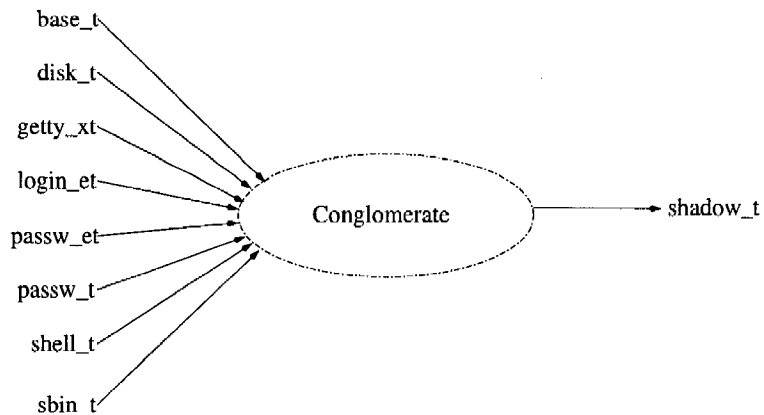


Figure 7.9: Modified BLP less than relation for the password policy.

Now type `shadow_t` is strictly greater than all other types. This is because `shadow_t` may only be read by `passw_d`, which is listed as a secrecy exception. Consequently, during

calculation of the $MBLP \leq$ relation we say that no domains may read `shadow_t`. However, since we did not list any integrity rules regarding `shadow_t`, it was listed in the write set for `passwd`. Since `passwd` could read `lib_t`, while it could write `shadow_t`, the BLP calculation algorithm found that $lib_t \leq shadow_t$. Note that if we had used the following rule:

```
protect shadow_t except from passwd
```

then `shadow_t` would have been unrelated to all other types.

The modified BLP also relates `passwd_t` as strictly less than all types to which it is related. This is because, the only domain which may write to `passwd_t` is listed as a write exception, causing `passwd_t` not to be placed in its write set. The type is therefore not placed in any write sets, and therefore is never calculated to dominate any other types. The combination of the `passwd_t` protection rule, and the resulting BLP relation, provide us more information about who may write to `passwd_t` than we would have either using straight BLP, or by listing `passwd` as a trusted domain.

Chapter 6 provided straightforward tools for editing and viewing DTE policies. This chapter presented more intricate techniques for policy analysis. The next chapter will similarly expand upon the creation of policies. The work of this chapter will also become more practical as it is integrated into the process of policy composition.

Chapter 8

Construction of DTE Policies from Modules

Chapter 6 presents tools to create, edit, and analyze DTE policies. However, when working with large policies, patterns begin to emerge. Policies typically consist of several sets of domains and types. The entities within a set work together to achieve some goal. However, the sets often interact very little. For instance, in the ftp policy presented in Figure 3.2, the domain `ftpd_d`, and the types `ftpd_t` and `ftpd_xt`, work together to protect the system from an unsafe binary. By removing these entities, and all references to them, the remaining policy becomes simpler. We will call this collection of domains, types, and all access rules pertaining to them, a module. The Ftp module is shown in Appendix A.2.3. The remaining base module is shown in Appendix A.2.1.

Allowing policies to be composed from simple, meaningful, and coherent pieces, will serve several purposes. First, creation of policies will become far more efficient. For instance, when adding a new domain to an existing policy, one might have to enter hundreds of type accesses in order to get it properly interacting with the current policy. In contrast, modules allow domains and types to be grouped at several levels, and access to be specified using

any of these groups.

Second, adding a feature to a policy, such as a new method of controlling access to the shadow file, or protection from a critical binary in which an as-yet unsolved vulnerability has been found, will become a simpler task. The module can be written entirely from its own point of view. Furthermore, in researching the state of the current policy, in order to understand how to properly insert a new feature, one need only look at those modules which can affect the new functionality.

Third, modules may be helpful in simplifying the analysis, and proof of invariants, of policies. For instance, several modules may be trivially shown to be irrelevant to the ability of the `inetd` daemon, if remotely exploited, to erase the `utmp` log file.

Finally, because a module generally encodes domains, types, and access rules which work together toward some end, it is a natural way to express the security policy changes necessary for a new piece of software. Software companies and free software groups, therefore, could distribute policy modules along with software packages.

We begin by describing the behavior of the DTE Policy Compiler (`dpc`), which we have written to construct a policy from modules. Next, we describe in detail, and prove the correctness of, methods to automatically ensure certain properties will maintained after module application. Finally, in order to show the usefulness of this idea, we will present several modules which, while simple and clean in themselves, will compose into a very powerful policy.

8.1 Policy Compiler File Formats

This section provides syntax specifications of the dpc control file, modules files, and patch files.

8.1.1 Control File Specification

The DTE policy compiler takes its instructions from a single control file. By default, this file is called `control`, although another file can be named on the command line. The sample control file shown in Figure 8.1 applies the Ftp module and a set of service modules to the base modules. In this section, we provide the BNF specification of the dpc control file, interspersed with further explanations.

```
<control_file> ::= <command_line> *  
  
<command_line> ::= <read_cmd> |  
                    <load_policy_cmd> |  
                    <apply_cmd> |  
                    <patch_cmd> |  
                    <write_cmd>  
  
<load_policy_cmd> ::= load_policy <policy_module> <policy_name>
```

The `load_policy` statement causes dpc to load a Policy Consistency Package, or pcp. Section 8.2 will describe this feature in detail. A pcp is implemented as a Perl module, meaning that the pcp named BLP must be located in the file `BLP.pm`. The `<policy_module>` is the name of the Perl module which implements the pcp. The `<policy_name` is the name

which dpc will assign to this pcp. The assertion statements discussed in Section 8.1.2 will associate themselves with pcp using this name.

```
<read_cmd> ::= read <file_glob>
```

This causes dpc to read a module file or set of module files from disk. Simple filename globbing is supported. For instance, line 3 in Figure 8.1 causes all files under directory `Service` to be read. A module file may contain any number of modules, none of which need reflect the filename. Each module is stored separately in memory. They are not yet combined.

```
<apply_cmd> ::= apply <module_glob>
```

Instructs dpc to apply a specified module or list of modules to the current policy. Applying a module means that its types and domains are added to the policy, and access rules between the types and domains of all applied modules are resolved. The algorithm for doing so is discussed in detail in Section 8.1.2. The modules must have previously been read using the `read` command. Modules may be specified by module name, or by namespace hierarchy. The namespace hierarchy is delimited by “.”. Appending “*” to a module name indicates all modules at this level level should be applied, while a “+” indicates all descendants must be applied. For instance, line 4 of Figure 8.1 would cause modules named “System.base.types” and “System.base.domains” to be applied, but not “System.base.security.passw”. Line 6, however, would cause “Service.daemons.sshd” as well as “Service.daemons.security.login.sshd” to be applied.

All modules specified in a single `apply` directive are applied simultaneously. See the discussion of domain and type grouping in Section 8.1.2 for details of how this affects the timing and, therefore, the results of group expansion.

```
<patch_cmd>      ::= patch <file_name>
```

This feature allows patching the final policy with simple changes for the sake of policy testing. A patch file can specify addition or removal of type assignment statements, domain transition rules, domain signal rules, entry points, and domain to type access rules.

```
<write_cmd>      ::= write <file_name>
```

This instructs `dpc` to write the policy as calculated thus far to the specified filename. Specifying the name `stdout` directs `dpc` to write to standard output. Line 8 in Figure 8.1 causes `dpc` to write the policy to the file `dte.conf`. Any number of write commands may occur throughout the control file, allowing the saving of policies at various stages of module application.

The remaining lines complete the above definitions.

```
<file_glob>      ::= <file_exp> +
<file_exp>       ::= <pathname> | <pathname>.*"
<module_glob>    ::= <module_exp> +
<module_exp>     ::= <module_name> | <module_name>.*" |
                    module_name.*"
# In other words, these are basic strings:
<module_name>    ::= [a-zA-Z][a-zA-Z\_-0-9.]+
```

```

<path-name>      ::= <file_name> [ / <file_name> ]*

<file_name>      ::= [a-zA-Z][a-zA-Z\_0-9.]+

<policy_name>    ::= [a-zA-Z][a-zA-Z\_0-9.]+

<policy_module> ::= [a-zA-Z][a-zA-Z\_0-9.]+

01. load_policy blp_mod blp
02. read System/Base
03. read Service/*
04. apply System.base.*
05. apply Service.ftp
06. apply Service.daemons.+
07. patch ftp.assert
08. write dte.conf

```

Figure 8.1: Sample dpc control file.

8.1.2 Module File Specification

Here we discuss the structure of a module file. We will use the Ftp module shown in Appendix A.2.3 as an example. The module syntax specification follows.

```

<module_file>   ::= <module>+

<module>        ::= Module <mod_name>

                  [<domain_def>|<type_def>|<group_def>]+

                  end

<domain_def>    ::= domain <dom_name>

                  <dom_line>+

```



```
end
```

A module file may contain more than one module. Each module may contain several domain, type, and group definitions, as well as the access rules pertaining to them.

```
<dom_line> ::= entries <type_name>+ |
              [absolute] signal [in|out] <gen_dom> <sig_num> |
              [absolute] domain [in|out] <gen_dom> [auto|exec|none] |
              [absolute] type <gen_type> <type_acc> |
              assert <policy_name> <data>
              DEFAULT_DOMAIN
```

The domain definitions declare a (unique) name for the domain, a set of entry types, and a set of access rules pertaining to the new domain. Domain transition or signal access rules may be “in”, in which case they specify access from other domains to the new domain, or they may be “out”, defining access from the new domain to other domains. Since types are passive objects, which cannot themselves access other types or domains, the type access rules in a domain definition do not include the “in” or “out” keyword.

Exactly one domain definition applied to a policy must contain the keyword “DEFAULT.DOMAIN”. That domain will be assigned to the first process on the system.

```
<type_def> ::= type <type_name>
              <type_line>+
              end
```

```

<type_line> ::= <path_type> <path_name>+ |
               [absolute] access <gen_dom> <type_acc> |
               <default_type> |
               assert <policy_name> <data>

```

```

<default_type> ::= DEFAULT_ETYPE | DEFAULT_UTYPE | DEFAULT_RTYPE

```

Type definitions declare a (unique) name for the type, a set of paths assignment rules, and a set of access rules. Clearly, the access rules are only incoming from domains. Either one type must also be associated with the “DEFAULT_RTYPE” keyword, or both the “DEFAULT_ETYPE” and “DEFAULT_UTYPE” keywords must be associated with one type each, in order to define default types for the file system.

Both type and domain definitions may contain “assert” statements. These are used for maintenance of policy constraints. They are stored with the type definition until module application, but their interpretation and enforcement is defined by the pcps as described in Section 8.2. The last line of the `ftpd_xt` type definition in the `Ftp` module is an example of an assert statement, instructing a module loaded as “blp” to label this type as protected.

```

<group_def> ::= group domain <dom_name>
               import <dom_name>+
               end

```

```

<group_def> ::= group type <type_name>
               import <type_name>+

```

end

`<gen_dom> ::= all | none | <dom_name>`

`<gen_type> ::= all | none | <type_name>`

Grouping is accomplished on several levels. First, one may simply specify “all” to refer to all domains or types which are currently known. Second, a group definition in a module may define a named group of domains or types. For instance, the module segment listed in Figure 8.3 defines a domain group consisting of several domains which are not trusted. The module segment listed in Figure 8.4 defines a type which is actually called `root_t`. Since this is the typename which will be used in the final DTE policy, no names within the namespace may actually clash.¹ Modules may refer to this type using any of the following names:

1. `root_t`
2. `base.extraneous.root_t`
3. `all`
4. `base.extraneous.*`
5. `base.extraneous.+`
6. `base.+`

¹This could be worked around by automatically randomizing the name in the event of a clash, but this simply was not a great concern for this prototype.

In addition, any type groups which have imported this type can also be used to refer to this type.

The name `base.extraneous` may be a real type, or it may simply be a namespace placeholder, depending on whether any module defines a type by that name.

```
<type_name> ::= [a-zA-Z] [a-zA-Z0-9_]*
```

```
<dom_name> ::= [a-zA-Z] [a-zA-Z0-9_]*
```

```
<path_name> ::= [ /<file_name> ] +
```

```
<policy_name> ::= [a-zA-Z] [a-zA-Z0-9_]*
```

```
<data> ::= *
```

8.1.2.1 Priority of Access Rules

Since domains and types can declare conflicting access rules, we must clearly define the priority of access rules. Much thought has been given to the current priorities, which have been somewhat modified following experience with an earlier module compiler prototype. The priority takes the form of an integer between 1 and 8. The priority assigned to access rules is shown in Table 8.2.

If two rules exist pertaining to the access permitted from a domain to another domain or type, then the rules with the highest priority will be applied. For instance, the base module's definition of type `base_t` specifies that all domains have "absolute" access "rxld" to `base_t`. However, the `Ftp` module's definition of domain `ftpd.d` contains the statement:

Type of access	Priority level
Absolute in	8
Absolute out	7
Single destination in	6
Single destination out	5
Group in	4
Group out	3
Default ("all") in	2
Default ("all") out	1

Figure 8.2: Priorities of access rules

```
absolute type all none
```

Since an “absolute in” access rule has a higher priority than “absolute out”, `ftpd_d` will receive “`rxld`” access to `base_t`. Without this, it would not be able to access any other types, as it could not descend to the files of those types. Similarly, the types defined in the `Ftp` module must specify access from domain `ftpd_d` as “absolute”, as that is the only access which will override that listed in the specification for `ftpd_d`. On the other hand, the specification for type `bin_t` includes a normal “group in” definition. As this is of a lower priority than “absolute out”, the access rule specified by the `Ftp` module is chosen, denying `ftpd_d` all access to type `bin_t`. This is a crucial element of the `Ftpd` module, preventing the `ftp` server from providing attackers with root shells, for instance.

Note that incoming access always overrides outgoing access. More specific rules override more general rules, unless the “absolute” keyword is present in one of the rules.

8.1.2.2 Group Expansion

As explained in Section 8.1, the module compiler applies sets of modules when directed to do so by the control file. Since more modules can be read later, we must clearly define the behavior of group expansion.

For named domain and type groups, the group is simply expanded at the time of module application. If the group has not yet been defined, an error is raised and compilation fails. For namespace globbing, that is, `*` and `+`, the currently defined descendants and children (respectively) of the parent being expanded are used. For instance, assume we applying a module which contains the rule

```
domain some_domain

    type base.exec.+ rwx

end
```

If the only children of `base.exec` defined thus far are `base.exec.sbin` and `base.exec.bin`, then only these types are included in this rule. A later module may define type `base.exec.javabin`, but this type will not be added to the access rule.

The `all` group behaves differently, however. An access rule directed at `all` will be expanded at the time of module application. However, a generic form of the rule is also stored. All such generic rules are expanded each time a set of modules is applied. If the rule had not previously been applied, any policy consistency modules will be consulted at the new rule creation, just as with any other new access rule. For example, the base module defines default access “`rld`” to type `base_t` for `all` domains. This rule is expanded after each module application, so that all domains will be granted this access.

8.1.2.3 Inheritance

An issue which may deserve further consideration is that of inheritance. It would seem to make sense to construct the type namespace such that certain properties, perhaps “absolute”

access rules, are automatically inherited by the children of a type. On the other hand, this may simply needlessly complicate the process of policy creation, the simplification of which is the precise goal of the policy compiler.

8.1.3 Patch File Specification

The patch file may specify any access rules, entry point, or type assignments which must, or must not, be a part of the final policy, regardless of the result of module application. The patch file is read and applied at the point where the control file instructs dpc to do so. Presumably, this would usually be the last action prior to writing the final policy. A sample patch file is shown in Figure 8.5. The syntax of the patch file follows.

```

<patch_file>      ::= <patch_line>*
<patch_line>     ::= [not] <patch_cmd>
<patch_cmd>      ::= domain_ep <domain> <type> |
                   domain_type <domain> <type_acc> "to" <type> |
                   domain_trans <domain> <x_acc> "to" <domain> |
                   domain_sig <domain> <signal> "to" <domain> |
                   type_assign <type> <ta_option> <path_name>

```

All aspects of the policy can be controlled by the patch file. No group expansion of any type is performed, so only plain domain and type names may be used. The meaning of a statement may be inverted by prepending with the word “not”. For instance, line 4 of Figure 8.5 indicates that the domain `ftpd_d` may not have read access to type `shadow_t`. If this access was granted by some module, then the access will be revoked.

In the case of a violation of a type assignment assertion, only a warning is given. This is because there is no single correct way to handle the violation. For instance, in the case of line 5 of Figure 8.5, if `/etc/passwd` is assigned the etype `user_t` as a result of a rule assigning `user_t` recursively to `/etc`, it is not clear whether `dpc` should revoke the original rule, or simply issue a conflicting rule for only the file `/etc/passwd`.

The remainder of the patch file specification follows.

```

<x_acc>          ::= "auto" | "exec" | "none"

<type_acc>       ::= "none" | [rwxlcda]+

<signal>        ::= [0-32] | "none"

<ta_option>     ::= "-r" | "-u" | "-e" | "-eu"

<domain>        ::= [a-zA-z][a-zA-Z_0-9]*

<type>          ::= [a-zA-z][a-zA-Z_0-9]*

<path_name>     ::= <file_name> [ / <file_name> ]*

<file_name>     ::= [a-zA-Z][a-zA-Z\_\_0-9.]+

group domain untrusted_domain_group
    import ftpd_d talkd_d fingerd_d
end

```

Figure 8.3: A group declaration combining some untrusted domains

```

type base.extraneous.root_t
    DEFAULT_RTYPE
    [...]
end

```

Figure 8.4: A possible definition of type `root_t`.


```

01. domain_type ftpd_d none to sbin_t
02. domain_type ftpd_d none to bin_t
03. domain_type ftpd_d rld to base_t
04. not domain_type ftpd_d r to shadow_t
05. not type_assign user_t -e /etc/passwd

```

Figure 8.5: A sample dpc patch file.

8.2 Automatic Maintenance of Policy Constraints

In chapter 7, we analyze DTE policies by using concepts from simpler access control systems to introduce relations on objects and subjects. TIS used this concept to enforce the maintenance of specific relations across applications of run-time policy changes. We generalize this idea by building into dpc a pluggable architecture to support maintenance of policy constraints for any policy.

Using the `load_policy` directive in the control file, the policy compiler is instructed to load a Policy Consistency Package, or pcp. The pcp is a generic Perl module. For instance, in the line

```
load_policy BLP blp
```

‘BLP’ is the name of the file (minus `.pm` extension) wherein the package is located, and ‘blp’ is the keyword which DTE modules will use to identify the module.

Recall that the dpc applies modules when requested by a line in the control file such as

```
apply Service.ftpd Users.*
```

All modules specified in the ‘apply’ line will be applied simultaneously. For any type or domain introduced in some module, the pcp will make consistency guarantees for all subsequent module applications. The pcp must define two functions, `pre_apply` and `post_apply`.

- `pre_apply` is called before the domains and types defined in the modules about to be applied are added to the global lists of domains and types. The `pcp` can therefore calculate the security properties, levels, or relations which exist and must be maintained.
- `post_apply` is called after the modules have been fully applied. The `pcp` may now decide whether any security properties which existed at `pre_apply` have been violated, and act accordingly. Since the `pcp` is a full Perl module, it can choose to simply warn of the violation, or stop the policy compiler altogether.

A `pcp` can thus be written to calculate and maintain any type of relation or property found to exist prior to module application. Obvious examples include the BLP \leq relation, the Ring policy \leq relation, assured pipelines, and the MBLP \leq relation. To provide further information to a `pcp`, modules may use the “assert” keyword as described in Section 8.1.2. The module syntax specifies that assert statements should include the `pcp` name followed by any data. The entire assert statement is stored with the rest of a module’s information for the domain or type to which the assertion relates, so that the `pcp` may find all assertions relevant to it during `pre_apply` and `post_apply`. The BLP `pcp` does not make use of “assert”. The `pcp` to enforce the modified BLP presented in Section 7.3 uses “assert” to append `secret` and `protect` statements with types and domains.

8.2.1 Correctness of the BLP PCP

We have implemented a sample `pcp` to enforce the maintenance of the BLP *-property. The main code of the `pcp` is shown in Appendix A.3.2, A.3.3, and A.3.4. We now show that

BLP.pm will in fact detect any violations of the BLP *-property. Recall that the *-property states:

$$\forall x \in \mathcal{D} \text{ and } a, b \in T : r(x, a) \wedge w(x, b) \Rightarrow L(a) \leq L(b)$$

We define \mathcal{D} and T as the sets of domains and types known before module application, and \mathcal{D}' and T' as the same sets after module application. We must show that `pre_apply` correctly calculates the \leq partial relation as defined by the *-property. Then we will show that, for all types in T , any changes to the \leq relations which are introduced by the module applications will be detected.

Theorem 8.1 *BLP::calculate_blp correctly computes the BLP \leq partial relation.*

Proof:

We will show this by contradiction. Let us assume that some domain y has read access to a type c , and write access to a type d , but `calculate_blp` does not report $c \leq d$.

`Calculate_blp` iterates over all domains in \mathcal{D} . Therefore, the loop spanning lines 04 to 17 would be entered once with `$dom` set to y . Since we have said that y has read access to c , the hash table `$domain->{"realta"}` exists, and contains the entry `{"y", "c"}`. We therefore will enter the loop begun on line 08 with `$type1` set to c . `$value` will contain the string representing access from y to c , which we have said contains 'r'. We therefore begin the loop on line 11. Since we know y to have write ('w') access to "d", we will reach line 14, with `$type1=c` and `$type2=d`. This contradicts our assertion. `%BLP::leq` does contain `{"y", "d"}`.

We must also show that $\%BLP::leq$ does not contain any pairs $\{“u”, “v”\}$, where in fact it is not the case that $L(u) \leq L(v)$. Let us assume that this did in fact occur. This could only happen if there exists a domain “y” such that $\%main::domains\{“y”\} \rightarrow \{“realta”\} \rightarrow \{“u”\}$ contains “r”, and $\%main::domains\{“y”\} \rightarrow \{“realta”\} \rightarrow \{“v”\}$ contains “w”. We will claim without proof that this could occur only if in fact domain “y” had read access to type “u”, and write access to type “v”. But if this is the case, then, by the BLP *-property, $L(u) \leq L(v)$.

■

Theorem 8.2 *At $BLP::pre_apply$, BLP calculates the correct \leq relation*

Proof:

$BLP::pre_apply$ is shown in Appendix A.3.3. Let us assume that there exists $y \in \mathcal{D}$ and $c, d \in \mathcal{T}$ such that $r(y, c) \wedge w(y, d)$, but $\%BLP::leq$ does not contain $\{“c”, “d”\}$.

$BLP::pre_apply$ calls $BLP::calculate_blp$. Therefore, if $\%BLP::leq$ does not contain $\{“c”, “d”\}$, then $BLP::calculate_blp$ did not correctly calculate the $BLP \leq$ partial relation. However, we have shown that it does in fact correctly calculate \leq .

We can use the same argument to show that if $\%BLP::leq$ contains $\{“c”, “d”\}$, then it must be that $L(c) \leq L(d)$.

■

Theorem 8.3 *If the $BLP \leq$ partial relation between the types in \mathcal{T} changes after module application, then the changes will be reported in line 13 of $BLP::post_apply$.*

Proof:

We will prove this by contradiction. `BLP::post_apply` is shown in Appendix A.3.4. Let us assume that the application of a module introduces a new relation $e \leq f$ on two preexisting types e and f . We will attempt to show that `BLP::post_apply` will not come to line 13 with `($a="e", $b="f")`, that is, it will not warn of this change in the `BLP ≤` partial relation.

Line 04 computes `%post_leq` using `BLP::calculate_blp`. As a consequence of our previous proof, we know `%post_leq` to correctly represent the `BLP ≤` partial relation. Therefore, `{"e", "f"} ∈ %post_leq`, and the loop beginning at line 05 will be entered with `$a="e"`. Since $e \in \mathcal{T}$, we will pass line 06. Line 07 will set `$b="f"`. Again, since $f \in \mathcal{T}$, we will pass line 08. Finally, since $e \neq f$, we will pass line 09.

By our previous argument, the fact that $L(e) \not\leq L(f)$ at `BLP::pre_apply` means that `("e", "f") ∉ %BLP::leq`. We state without proof that the the function `path_exists_orig` shown in Appendix A.3.1 is correct. It therefore will detect that `("e", "f") ∉ %BLP::leq`, and that there exists no $(t_1, \dots, t_n) \in \mathcal{T}$ such that

$$(e, t_1), (t_1, t_2), \dots, (t_{n-1}, t_n), (t_n, f) \in \%BLP :: leq$$

That is, there exists no set of types related by \leq such that, by transitivity, $e \leq f$. The function `path_exists_orig` therefore return false. Therefore, line 13 of `BLP::post_apply` will be reached with `$a="e"` and `$b="f"`, generating a warning about the change in the `BLP ≤` partial relation.

■

8.2.2 Modified BLP PCP

The MBLP policy is introduced in section 7.3. We have coded a `pcp` to implement MBLP. Relevant excerpts of the code are shown in Appendix A.4.

We wish to show that this `pcp` correctly detects changes in the $MBLP \leq$ relation. As in the BLP `pcp` correctness proof, we begin by showing that `calculate_mblp` in fact computes the correct $MBLP \leq$ relation on all types. A precise specification of the relation is given in Equation 7.1.

Theorem 8.4 *The `calculate_mblp` function shown in Appendix A.4.1 correctly calculates the $MBLP \leq$ relation.*

Proof: Before calling `calculate_mblp` on line 5, `pre_apply` calls `setup_asserts` on line 4. This function walks through all MBLP assert statements which have been read. These assert statements specify secret types, protected types, trusted (ignored) domains and types, and secrecy and protection exceptions. As these lists are stored as simple arrays, we state without proof that each of the above functions returns the appropriate boolean:

1. `is_ignore_type(type)`
2. `is_ignore_domain(domain)`
3. `is_secret_type(type)`
4. `is_protected_type(type)`
5. `is_protect_exception(type, domain)`
6. `is_secret_exception(type, domain)`

The MBLP is defined in Equations 7.1 through 7.3. We begin by showing that, if the relation

$$t_1 \leq t_2$$

is defined by the `calculate_mblp` function, then it is in fact the case that $t_1 \leq t_2$. The definition of this relation occurs at line 27.

```
27:          append_leq(%leq, $type1, $type2);
```

At this point, the variable `$dom` represents the domain d , the variable `$type1` represents the type t_1 to which d has read access, and the variable `$type2` represents the type d to which d has write access. To reach line 27 with each of these variables so set, each of the following must be true:

1. d is not a trusted domain. Else this domain would be skipped at line 07.
2. d has read access to t_1 . Else this type would be skipped at line 11.
3. t_1 must not be an ignored type. Else this type would be skipped at line 12.
4. d is not listed as a secrecy exception for t_1 . Else this type would be skipped at line 13.
5. d has write access to t_2 . Else this type would be skipped at line 20.
6. t_2 is not an ignored type. Else this type would be skipped at line 21.
7. d is not a write exception for t_2 . Else this type would be skipped at line 22.

Conditions 1-4 satisfy each condition in Equation 7.2, while conditions one and 5-7 satisfy Equation 7.3. This combination satisfies the left hand side of Equation 7.1. Therefore, if `calculate_mblp` defines $t_1 \leq t_2$ for some policy, then this relation holds under MBLP.

We next show that if it is the case that $t_1 \leq t_2$, then the `calculate_mblp` will define $t_1 \leq t_2$. If $t_1 \leq t_2$, then by Equations 7.1 through 7.3, there must be some domain d such that $mr(d, t_1)$ and $mw(d, t_2)$. By the definitions of $mr(d, t_1)$ and $mw(d, t_2)$, the domain must not be a trusted domain, t_1 and t_2 must not be ignored types, and d must not be a read exception for t_1 or a write exception for t_2 . Looking back to the `calculate_mblp` function in Appendix A.4.1, we see the main loop spanning lines 4 to 30 iterates over all domains. Lines 6 and 7 skip a domain only if d is a trusted domain or has no permitted type accesses at all. Otherwise we reach the nested loop which begins with line 9, iterating over all types to which d has some access. Types to which d does not have read access, or which are ignored types, are skipped on lines 11 and 12. Line 13 skips to the next type if d is a secrecy exception for the current type. Lines 18 through 28 iterate again over all types to which d has access. Line 20 limits the loop to types to which d has write access, and line 21 skips types for which d is a write exception. Note that each case where a type has been skipped has corresponded to an exception in Equations 7.2 or 7.3. In all other cases, we reach line 27, which defines the relation $t_1 \leq t_2$.

The function `calculate_mblp` therefore correctly calculates the $\text{MBLP} \leq$ relation for any DTE policy.

■

Theorem 8.5 *If a module \mathcal{M} adds a new MBLP \leq relation between two preexisting types, then `post_apply` will warn of this new relation.*

Proof: We have shown in Theorem 8.4 that `calculate_mblp` correctly calculates the MBLP \leq relation. Line 05 of `pre_apply` places the \leq relation into the variable `%leq`. As this is called before module application, `%leq` contains exactly the MBLP \leq relation prior to application of module \mathcal{M} . Line 04 of `post_apply` places the result of `calculate_mblp` into the variable `%post_leq`. As this is called after application of module \mathcal{M} , `%post_leq` contains exactly the MBLP \leq relation after application of module \mathcal{M} .

Assume module \mathcal{M} added a relation $a \leq b$, where prior to application of \mathcal{M} , $a \not\leq b$. This means that at line 09 in `post_apply`, `%post_leq(a)` does contain b , while `%leq(a)` does not. Through the nested loops spanning lines 07-18 and 09-17, each pair of base types (a, b) for which `%post_leq` defines $a \leq b$ will be checked. Since `%leq` does not also define $a \leq b$, line 17 will warn of the new relation. Since, again, we have shown `%post_leq` and `%leq` to be correct, this will be the case if and only if the module \mathcal{M} introduced this relation, while the policy prior to application of \mathcal{M} did not contain this relation. ■

In Sections 7.3.1 and 7.3.2 we showed that the modified BLP can enforce assured pipelines and Clark-Wilson CDIs. Since we have just shown that the MBLP `pcp` correctly enforces the maintenance of modified BLP properties across module application, it follows that the MBLP `pcp` can be used to enforce assured pipelines and CDIs. Of course, other policy consistency packages can be written to enforce any access control policies desired by the security administrator. This is the most significant contribution of policy compiler's `pcp` architecture.

8.3 Sample Modules

Previous sections have argued for the usefulness of modules, and presented their syntax. This section will present several modules which we have used, to the same ends. We begin with a base module, which defines types and domains which will be used by all other modules. Next we present a module to introduce a password domain, which can be used by ordinary users to change their passwords, a task requiring permissions which ordinary users lack. Following is an Ftp module, implementing protections similar to those of the policy presented in [27]. Finally, we present a module to implement an assured pipeline for the system log daemon.

8.3.1 Base Module

The feasibility of specifying simple but powerful modules in order to enhance a DTE policy will depend on the ability to specify an appropriate base module. We desire little or no namespace clashing between modules, but also wish to keep modules concise, with a minimum number of access rules. Therefore we want very few, well thought out domains. We want to avoid having too many types, but at the same time we want to split up files which have a system-wide meaning. For instance, it is not our place to assign types to the `/etc/sshd_config` and related files. However, it is better that we take care of `/lib` and `/usr/lib` now, since there is not one single module which can lay claim to those. A base module which we believe satisfies these subjective criteria is shown in Appendix A.2.1.

The domain groups under `Admin` are undefined in the base module, but are intended to be defined in later modules, either as domains or groups of domains. For instance, each

could be defined as a group containing the `root_d` domain, although this would result in a far less secure system than this base module is attempting to produce.

8.3.2 Password Module

The base module defines types `passwd_t` and `shadow_t`, and assigns type `passwd_t` to file `/etc/passwd` and `shadow_t` to `/etc/shadow`. However, it does little with these types. Users may not change their passwords, for instance, as they are simply denied all access to `shadow_t`.

The `password` module shown in Appendix A.2.2 defines a new domain, `passwd_d`, which may be entered through `/usr/bin/passwd` by any domains defined in the base module. Type `passwd_t` is redefined to include a lock file and the temporary file `/etc/passwd.tmp`. It may now be read by all domains, since anyone is welcome to basic user information. However, it may be written only by `passwd_d`. Type `shadow_t` is also redefined. All domains are denied any access to it. The only exceptions are `login_d`, which may read the shadow file, in order to verify users logging in, and `passwd_d`, which may write `shadow_t` in order to change passwords. Note that, under this policy, a successful attack against most daemons running on the system will still not allow the attacker to read even encrypted passwords. So long as we write a simple, and secure, `passwd` program, this module affords the same flexibility as any current Unix system, combined with far greater security.

8.3.3 Ftp Module

The `ftp` module is a purely restrictive one. It is designed to allow anonymous ftp to be offered despite known and unpatched vulnerabilities in the ftp daemon. It first denies all

domains, except the newly defined `ftpd_d` domain, execute access to `/usr/sbin/in.ftpd`. It next prevents `ftpd_d` transitioning into any other domains, and denies it execute access to any but its own executables, located under `/home/ftp/bin`, its entry point, and system libraries. It is therefore impossible, for instance, to offer a root shell, since `ftpd_d` may not execute a shell. Write access is very strictly guarded as well. Since `ftpd_d` may not execute any types which it may modify, there is no possibility of any Trojan horse attacks. The `ftp` module is shown in Appendix A.2.3.

8.3.4 Syslog

The `syslog` module, shown in Appendix A.2.4, implements an assured pipeline (See Chapter 2.2.5) as an ideal setting for Bruce Schneier's secure logging scheme [43]. No logging algorithm can in itself prevent an attacker from forging new log entries. This algorithm, however, facilitates the detection of modification or deletion of log entries which were committed before the system compromise. The hope is that one of the committed log entries will warn of the attack in progress. In combining this algorithm, an assured pipeline, and the enhanced protection of DTE over all system services, we aim for these approaches to complement each other, providing a more secure system than either could provide separately.

8.3.5 Control File

The module compiler must be directed by a control file. For the sake of completeness, a control file directing `dpc` to create a DTE policy from all the modules listed above follows.

```
read base.module  
  
apply System.Base  
  
read passwd.mod  
  
read ftp.mod  
  
read syslog.mod  
  
apply Service.*  
  
write stdout  
  
write dte.conf
```

8.4 Conclusion

This chapter presented a method and tools for creating security policies from modules. The tool provides support for enforcement of arbitrary policy assertions or relations between domains and types. We believe the result is a system which is at the same time simple, powerful, and flexible enough to permit collaboration in construction of security policies. By permitting a community to standardize upon a generic base module and type and domain namespace hierarchies, it becomes possible for a new application to be distributed with a policy module which can be integrated into end user policies. The work presented in this chapter is therefore essential to the end goal, stated in Chapter 1, of providing both easy to use and simple to administer mandatory access control for Linux.

Chapter 9

Implementation Extensions

The current DTE implementation is sufficient to set up a flexible and secure MAC system. However, several extensions would far increase its usability. We discuss these here, and provide implementation details. However, these extensions have not been implemented.

The first extension copes with a new Linux file system feature, namespaces. The second provides a new method for assigning types to files, which is useful in particular for the `/proc` file system.

Finally, we describe not a kernel extension, but a method for extending DTE protections to network services. We use the traditional example of an NFS server.

9.1 Namespaces

A new, thus far little used feature in Linux, is that of per-process namespaces [52], a concept first introduced in the Plan 9 operating system [40]. Traditionally, all processes in a Unix system see the same file system tree. Initially, the same is true under Linux. A new option to the `fork` system call requests the cloning of a new file system tree. The forked process then receives a copy of the tree, cloned recursively starting from the file system root. Any

changes effected by mount activity in the new tree are seen only by the forked process and its descendants. Likewise, changes to the old tree are not seen by processes using the new tree.

9.1.1 Problem

The file system clone is implemented by a recursive copy of `vfsmount` structures. These kernel structures are the glue which holds file system trees together, and are shown in Figure 2.7. For instance, a mount of `/dev/hda3` under `/mnt/misc` would create a `vfsmount` structure with pointers to the root dentry for the file system stored on `/dev/hda3`, as well as its superblock, and insert this structure at the `d_vfsmnt` pointer of the dentry for `/mnt/misc`. If `/dev/hda3` is already mounted under `/mnt/disk`, for instance, a mount of `/mnt/disk1/scratch/d1` would result in a similar `vfsmount`, but with its root dentry pointer set to the `scratch/d1` directory on that disk.

In Chapter 3, we show that the malleability introduced by binding is resolved by using information stored at the superblock. A “real” mount location for a file system points to a `vfsmount/dentry` pair, which together specify a single location in the file system tree. We refer the real mount location of a file system as the “real parent” of the root of the mounted file system’s root directory. When using multiple namespaces, the real mount location for a file system may point to a `vfsmount` not in the current process’ namespace. At first glance this feature seems safe to ignore. However, it becomes dangerous when processes exit and namespaces are destroyed. The following scenario is not handled by the current DTE implementation:

0: Process 1	Process 2
1: <code>fork(clone, ...)</code>	
2:	<code>(begin under new namespace)</code>
3: ...	<code>mount /dev/hda3 /mnt/b</code>
4:	<code>exit</code>
5: <code>ls /mnt/b</code>	

At line 5, the lookup of `/mnt/b` will cause the kernel to dereference the real parent `vfsmount` for `/mnt/b`, which existed under the namespace for process 2. Since this namespace no longer exists, the reference is invalid.

9.1.2 Solution

The problem introduced by per-process namespaces could be dealt with in several ways. We will mention two. The first is to ensure that all real parent `vfsmount` structures are located in the original namespace, and that this namespace is never deleted. The second, simpler solution would be to increase the namespace usage counter each time one of its `vfsmounts` is referenced by a superblock's real parent pointer. This prevents the namespace being deleted until all such superblocks are freed.

Both of these solutions permit the crossing of namespaces during a `dte_d_path` call, which is the only function using the real parent pointers. However, since throughout all mount activity, DTE maintains information permitting it to recall the original tree structure, as described in Section 3.3, this will cause no ill effects. The additional code required to cope with namespaces is shown in Figures 9.1 through 9.3.


```

=====
File: security/dte/read_policy.c
Beginning at line: 1543
=====

int read_dte_config(void)
{
    struct dte_fdata stat;
    char *c;
    int line = 0;
    int err = 0;

    /* initialization of variables and memcaches */
    num_dte_domains = 0;

+     dte_root_ns = current->namespace;
+     atomic_inc(&namespace->count);
+     dte_root_dentry = dget(current->fs->root);
+     dte_root_mnt     = mntget(current->fs->rootmnt);

    dte_domains = kmalloc(8192,GFP_KERNEL);
    dte_type_cache = kmem_cache_create("dte_type_names",4096,
        0,0,NULL,NULL);
    dte_path_cache = kmem_cache_create("dte_path_names",4096,
        0,0,NULL,NULL);

```

Figure 9.1: Modification to DTE setup to store root namespace and prevent its unloading.

9.2 Accommodating proc

The policy analysis sections (Sections 6 and 8) demonstrated that in order to allow powerful constructs such as CDIs and assured pipelines, it must be possible to segment domains. This means that they are not forced to read and write types which other domains may access. There is nothing inherent in Unix systems to prevent this. Executables to be run by a segmented domain may be compiled statically, eliminating the need to access shared library types, and may access their own private /tmp directories. Likewise access to devices need only be carefully planned in advance. However, Linux does make it impossible to segment

```

=====
File: security/dte/mount.c
Beginning at line: 188
=====

/* This function looks a pathname up much as the plain path_lookup,
   but always starts from the root directory of the root namespace */
int dte_path_lookup(const char *name, unsigned int flags,
                   struct nameidata *nd)
{
    nd->last_type = LAST_ROOT; /* if there are only slashes... */
    nd->flags = flags;
    spin_lock(&dcache_lock);

    /* Because of how we are called, the pathname must
       * always be absolute! */
    nd->mnt = dte_root_mnt;
    nd->dentry = dte_root_dentry;
    nd->old_mnt = NULL;
    nd->old_dentry = NULL;
    current->total_link_count = 0;
    return link_path_walk(name, nd);
}

```

Figure 9.2: New DTE function to descend pathname using root namespace.

a domain using purely static policies. The problem is that domains must have access to files under `/proc`, the pathnames to which cannot be predicted. If a process running under domain `user_d` has process id (`pid`), then its `proc` files will be located under the directory `/proc/(pid)/`.

The only way to handle this using purely static policies is to assign a common type to `/proc`, which all domains may read and write. However, the introduction of the BLP relation (see Section 7.2) on types will expose the problems this introduces. In brief, it provides a venue for domains to leak secret information to each other, as well as a source

```

=====
File: security/dte/mount.c
(...continued...)
=====

void hierarchical_setup(struct vfsmount *mnt)
{
    [ ... ]
    /* Now get an nd struct for the given pathname.
     * Note we do so using the root of the root namespace
    retval = dte_path_lookup(r->path,
        LOOKUP_FOLLOW|LOOKUP_DIRECTORY,
        &nd2);
    [ ... ]
    if (retval) {
        [ Log error message ]
    } else if (nd2.mnt != mnt) {
        [ Log the pretend mount ]
        /* Now we hook the pretend pathname - which, again,
         * came from the root namespace */
        sb_sec->mnt_parent = mntget(nd2.mnt);
        sb_sec->mountpoint = dget(nd2.dentry);
        dte_copy_ino_sec(nd2.dentry->d_inode, mnt->mnt_root->d_inode);
        path_release(&nd2);
        return;
    }
    [ ... ]
}

```

Figure 9.3: Modification to DTE hierarchical mount information setup.

of information with which to corrupt high integrity data.¹

Most DTE-like implementations take a much more dynamic approach to type-name binding than our implementation. Once a file system has been initialized using the type assignment rules of some policy, any files subsequently created are assigned typenames based upon the domain creating the file. Conceptually, this greatly complicates the state of a system after it has run for some time. One cannot actually predict the type which is

¹For instance, the data for a Trojan horse replacement for a critical binary

assigned to a file except immediately after system initialization.² However, for the `proc` file system, this approach provides a natural solution.

We are not willing to give up the clarity of static type-name binding which our implementation of DTE uses for permanent file systems. However, for one-time file systems, the more dynamic approach may often be the right one. Of course, since we currently see a case where an alternative is useful for a one-time file system, we cannot predict that we will not find such a case for a persistent file system. We therefore propose the following non-trivial enhancement to our implementation of DTE.

All domains will be associated with a default type, as is the case in other DTE-like systems. In most cases this will be ignored, and type assignment will continue to be based upon pathnames. However, a new form of mounting restriction will be allowed. In addition to specifying “pretend” mounting pathnames by device, as well as mount location restrictions, policies will be able to specify statements of the form

```
restrict [maj] [min] use_default_domain_type
```

or

```
restrict fstype use_default_domain_type
```

For instance, specifying

```
restrict proc use_default_domain_type
```

will give us our desired result.

²Note that this is *not* system boot. System initialization is a one-time event.

It must be noted that the `proc` deficiency is mainly one of analysis. Actually using `/proc` to directly leak information is infeasible, as no process is allowed write access under `/proc/(pid)`. Therefore `/proc` could at best be used as a subliminal channel [42].

9.3 Providing Network Security

Most existing TE and DTE implementations have extended their protections by sending domain and type information along with each network packet. We now consider in greater detail the justifications for doing so.

9.3.1 Security

In Section 3.2.5, we claim that extending DTE protection to network packets is not very helpful. Consider a domain name server. We might wish to assign some type, such as `auth.dns_t`, to all DNS query responses. This type name would indicate that the source of the packet was in fact a valid DNS server. In this way, any packets not of this type could be considered unsafe. However, subsequently claiming that packets which are of this type are safe, would be fallacious. Simply typing network packets does not provide any additional security. Forging this information is no more complicated than forging a packet to come from a different host, which would still be required were we not using DTE over the network.

In order to make the DTE type information worthwhile, it must be authenticated using a public key cryptosystem.³ An example protocol is shown in Figure 9.4. This protocol aims to authenticate the server, but takes no steps to authenticate the client to the server.

³No existing implementation of TE or DTE has gone to this effort.

We assume that dispensing DNS information does not violate any security policy. The DNS server possesses private key PR , and all clients have access to its corresponding public key PU . We use E_{PR} to indicate public key encryption, D_{PU} to indicate public key decryption, and C_K to indicate symmetric key encryption and decryption. Q is the DNS query.

Implementing this protocol at the kernel level would greatly complicate the DTE code, but would be worthwhile if it in fact provided additional security. In fact, there is no doubt that this protocol in itself provides a great deal of security. However, integrating DTE into this protocol gains us nothing. The addition of the `auth_dns.t` field into the responses by the DNS server provides no security beyond that afforded by the protocol itself. Therefore, the protocol might as well be implemented at the application level, or outside the DTE subsystem at the kernel level. The private key PR should, of course, be protected using DTE such that no service running on the DNS server, except the DNS service itself, is allowed to read, or modify the key. Now the key verifies that the DNS server is not being spoofed, and nothing but a compromised `named` itself could compromise the key. Of course, if `named` is compromised, nothing can be done, so this service also must be protected using DTE.

We have just shown that DTE cannot protect a client from a malicious server. Likewise, DTE cannot protect a server from a malicious client. An obvious example is an NFS server which exports file types along with the file system. While a friendly and DTE-aware system may respect the type assignments, a malicious client will simply ignore this information and freely publicize top secret data.

Finally, extending DTE over the network makes the synchronization of policies throughout the network far more important. For instance, an NFS server might export a file of

type `sbin.t`. It would seem prudent for the server to deny root write access to this type, which generally contains more important, and less frequently updated, types than `bin.t`. However, if any client is configured to permit root write access to `sbin.t`, perhaps for the sake of convenience, then the server's policy is immediately compromised. Provided that DTE control of networking does become a reality, policy synchronization may become an interesting new area of research, as the question of authenticating the source of updates, and minimizing the requisite trust placed in the subsystem performing the updates, appears quite complicated.

Step	DNS Server	DNS client
1	Create nonce R	
2	$P_1 = E_{K_{PR}}(R, 1, \text{"auth_dns_t"})$	
3	Send P_1	Receive P_1
4		$(R, X, T) = D_{K_{PU}}(P)$
5		Break unless $T = \text{"auth_dns_t"}$ and $X = 1$
6		$P_2 = C_R(Q, 2)$
7	Receive P_2	Send P_2
8	$(Q, X, 2) = C_R(P_2)$	
9	Break unless $X = 2$	
10	Calculate response as V	
11	$P_3 = C_R(V, 3, \text{"auth_dns_t"})$	
12	Send P_3	Receive P_3
13		$(V, X, T) = C_R(P_3)$
14		Break unless $T = \text{"auth_dns_t"}$ and $X = 3$
15		Use V

Figure 9.4: Protocol to provide server authentication

9.3.2 Convenience

We have shown above that using DTE to protect network data is not justified. Of course, there remains the possibility that DTE-controlled networking provides convenience. DTE would append two pieces of information to each network packet. The first is the type of the data, however that is determined. The second is the domain of the process which generated the data.

For many system services, these provide no additional information. Some services provide only one type of data, which is uniquely identified by the port over which the service is accessed. Examples include `ntpd`, which provides clock synchronization information, and `fingerd`, which provides user information.

The example which TIS provided of a useful feature supported by the DTE control of networking is for exporting DTE types along with files from a NFS server. In this section we discuss methods of integrating DTE and NFS.

9.3.2.1 Static Type Assignment

NFS can, of course, be used with our DTE prototype without any further extension. The simplest, and perhaps safest, method of doing so would be to assign an untrusted type, `external_t`, to a path `/nfs`, under which all file systems imported over NFS are mounted. The NFS server can be set up so as to prevent modification of files, while the client forbids execution of NFS-mounted files.

This protocol is not very useful. Files served by NFS need to appear to be a more natural part of the client file system. With a little planning, however, static type assignment to local pathnames can be used provide adequate support for NFS. For instance, if we

were to import `server:/export/usr/local` under `/nfs/usr/local`, we could assign an appropriate type, be that `bin_t` or `nfs_bin_t`, while another file system from the same server, `server:/export/home` could be mounted under `/nfs/home2`, and again assigned an appropriate type. This simple and obvious solution should in fact be sufficient, since NFS setups are in most cases very static. A file server may export some binaries in order to save space on clients, and some `/home` and data directory trees in order both to facilitate centralized backups, and to allow users access to their data from any client. Once the exports are in place, they rarely change, unless the file server itself undergoes a significant change as well, such as the installation of a new disk.

9.3.2.2 Server-Directed Type Assignment

TIS was not satisfied with this setup. They wanted to grant the NFS server the ability to export typenames along with files. In this way, an administrator could decide that a new directory, `server:/export/usr/local/sbin`, should be assigned `nfs_sbin_t`, and make the change only once, at the server. Perhaps more convincing, types created by users under `server:/export/home` would be exported under the appropriate type. TIS therefore typed all network packets, and created a DTE-aware NFS client which used this information to assign types to NFS files. In addition, it used the source domain attribute to file write requests from NFS clients in order to authenticate NFS writes. If we wish to provide this functionality, we have two options. First, we can, as TIS did, extend DTE functionality into the kernel networking code. Second, we can modify the NFS server and client code to provide this functionality with less additional kernel support.

The implementation which we propose will alter the NFS protocol to explicitly export

the type of a file along with a file itself. The NFS server now requires no expansion of DTE controls. The `nfsd` daemon is modified to add an `int dte_type` field to the `nfs_fattr` structure, which we append at the end of the `fs/nfsd/nfsxdr.c:encode_fattr` function, and decode at the end of the `fs/nfs/nfs2xdr.c:xdr_decode_fattr` function. The NFS client simply copies the `dte_type` field from the `svc_fh` structure into the LSM inode security field. These extensions are shown in Figures 9.5 through 9.8. In addition, at the start of the NFS mount, the server and client must agree on an index to typename conversion, which of course predisposes agreement on a set of typenames.

As a further extension of the `nfsd` daemon, it could be permitted to set types exported to clients differently than those on the server. For instance, the directory served from `server:/export/home` may be labeled `export_t` at the server, but `home_t` at the client. Since clients and servers frequently view the same file system differently, this should be a very useful feature.

The administrative convenience afforded by this architecture can be combined with increased security by using a two-way version of the public key cryptography protocol described in Figure 9.4. In that case the NFS server can refuse to serve any files, or only files of certain types, based upon the client's key, or lack thereof, and clients can be assured of the authenticity of the data they are served. Additionally, since adding the `dte_type` field to the `svc_fh` field amounts to a fundamental NFS incompatibility, the server must fall back to the standard protocol if it is to be allowed to serve non-DTE clients. This was not a problem for TIS, as they utilized the IP options field to share type information, which non-DTE machines would simply ignore. ⁴

⁴This is not quite true, as some operating systems crashed when presented with nonstandard IP options.

We have proposed an alternative architecture for NFS servers to share type information with DTE-aware clients. We argue in its favor on account of the simpler implementation - only a few lines of kernel code need be changed. However, TIS' solution is more general. Should there in fact turn out to be many services which benefit from the sharing of DTE types, then TIS' solution will be more appropriate. We, however, do not believe this to be the case, and feel that DTE is better used to protect binaries, libraries, and cryptographic keys, than to directly protect network protocols itself.

9.4 Conclusion

This chapter presented extensions to, and uses of, the DTE module whose implementation we feel would significantly enhance the module's usefulness. These extensions have not yet been implemented, but are considered a fruitful area of future work.

DTE network servers therefore did not send DTE information to such hosts.

```

=====
File: fs/nfsd/nfsxdr.c
Beginning at line: 134
=====

static inline u32 *
encode_fattr(struct svc_rqst *rqstp, u32 *p, struct svc_fh *fhp)
{

    [ ... ]

    if (rqstp->rq_reffh->fh_version == 1
        && rqstp->rq_reffh->fh_fsid_type == 1
        && (fhp->fh_export->ex_flags & NFSEXP_FSID))
        *p++ = htonl((u32) fhp->fh_export->ex_fsid);
    else
        *p++ = htonl((u32) stat.dev);
        *p++ = htonl((u32) stat.ino);
        sb_sec = (struct dte_inode_sec) stat.dentry->inode->i_security;
        *p++ = htonl((u32) sb_sec->etype);
        *p++ = htonl((u32) sb_sec->utype);
        *p++ = htonl((u32) stat.atime);
        *p++ = 0;
        *p++ = htonl((u32) lease_get_mtime(dentry->d_inode));
        *p++ = 0;
        *p++ = htonl((u32) stat.ctime);
        *p++ = 0;

    return p;
}

```

Figure 9.5: The code to export DTE types from NFS server.

```

=====
File: fs/nfs/nfs2xdr.c
Beginning at line: 98
=====

static u32 *
xdr_decode_fattr(u32 *p, struct nfs_fattr *fattr)
{
    [ ... ]
    fattr->du.nfs2.blocksize = ntohl(*p++);
    fattr->rdev = ntohl(*p++);
    fattr->du.nfs2.blocks = ntohl(*p++);
    fattr->fsid = ntohl(*p++);
    fattr->fileid = ntohl(*p++);
+   fattr->etype = ntohl(*p++);
+   fattr->utype = ntohl(*p++);
    p = xdr_decode_time(p, &fattr->atime);
    p = xdr_decode_time(p, &fattr->mtime);
    p = xdr_decode_time(p, &fattr->ctime);
    fattr->valid |= NFS_ATTR_FATTR;
    if (fattr->type == NFCHR && fattr->rdev == NFS2_FIFO_DEV) {
        fattr->type = NFFIFO;
        fattr->mode = (fattr->mode & ~S_IFMT) | S_IFIFO;
        fattr->rdev = 0;
    }
    return p;
}

```

Figure 9.6: Code to import DTE types into NFS client from the network.

```

=====
File: fs/nfs/inode.c
Beginning at line: 645
=====

/*
 * Look up the inode by super block and fattr->fileid.
 */
static struct inode *
__nfs_fhget(struct super_block *sb, struct nfs_fh *fh,
            struct nfs_fattr *fattr)
{
    [ ... ]

    if (inode->i_state & I_NEW) {

        [ ... ]

        inode->i_size = new_isize;
        inode->i_mode = fattr->mode;
        inode->i_nlink = fattr->nlink;
        inode->i_uid = fattr->uid;
        inode->i_gid = fattr->gid;
+       i_sec = (struct dte_inode_sec *)inode->i_security;
+       i_sec->etype = nfs_conv_dtetype(sb, fattr->etype);
+       i_sec->utype = nfs_conv_dtetype(sb, fattr->utype);

        [ ... ]
    } else
        nfs_refresh_inode(inode, fattr);

    [ ... ]
    out:
    return inode;

out_no_inode:
    printk("__nfs_fhget: iget failed\n");
    goto out;
}

```

Figure 9.7: NFS client code to copy DTE types into inodes at first read.

```

=====
File: fs/nfs/inode.c
Beginning at line: 997
=====

int
__nfs_refresh_inode(struct inode *inode, struct nfs_fattr *fattr)
{
    [ ... ]

    inode->i_mode = fattr->mode;
    inode->i_nlink = fattr->nlink;
    inode->i_uid = fattr->uid;
    inode->i_gid = fattr->gid;
+     i_sec = (struct dte_inode_sec *)inode->i_security;
+     i_sec->etype = nfs_conv_dtetype(sb, fattr->etype);
+     i_sec->utype = nfs_conv_dtetype(sb, fattr->utype);

    if (fattr->valid & NFS_ATTR_FATTR_V3) {
        /*
         * report the blocks in 512byte units
         */
        inode->i_blocks = nfs_calc_block_size(fattr->du.nfs3.used);
        inode->i_blksize = inode->i_sb->s_blocksize;

        [ ... ]
    }
}

```

Figure 9.8: NFS client code to insert DTE types into inodes on refresh.

Chapter 10

Conclusion

This work has presented an efficient, non intrusive implementation of Domain and Type Enforcement for the Linux kernel. It also presented user-level tools for the creation, analysis, and maintenance of DTE policies.

Parallel to this effort, a few other projects have also implemented various security extensions and mandatory access control systems for Linux. The most popular of these is the National Security Administration (NSA)'s SELinux. At the moment, SELinux supports Type Enforcement, on which DTE was based, as well as RBAC. Its RBAC implementation, however, is designed as a bridge between the Unix user concept, and TE domains. Most SELinux work is taking two forms. First, the security module itself is continually being improved. As it benefits from NSA employees being paid for working on SELinux full time, it has consistently been fastest out of all LSM modules to keep up with changes to the Linux kernel. Second, much work is being done to develop useful SELinux policies. Note that this is distinct from the work presented here. We provide tools and processes for analyzing and building policies. The SELinux project, without the benefit of such tools, is attempting to develop policies which both secure a Linux system, and minimize user impact. In other words, the work presented here should benefit the SELinux policy developers.

Clearly, a useful next step for this project would be to join efforts with the SELinux community. Extending the DTE Policy Composer (dpc) to output both DTE and SELinux TE policies should be straightforward. Further modifications to support SELinux RBAC could easily be implemented as a post-processing filter, such as dpc's "patch". We would also like to consider merging the two, such that a module can specify both TE and RBAC rules.

Another direction in which we plan to take this work, is to study and implement more pcps for dpc. A pcp can implement any type of algorithm which involves attaching information to a type or domain in a module definition, and using this information to observe and record properties before and after all subsequent module applications. The pcp could therefore enforce maintenance of powerful access control policies such as assured pipelines, or the modified BLP which our pcp enforces. It can also enforce arbitrary assertions, such as "No domain which may read type a.t, should ever be allowed to reach domain b.d." Further possibilities should emerge as we continue to study properties of both safe and unsafe policies.

Despite adapting our tools to work with SELinux policies, the DTE kernel module will continue to be maintained. There are several reasons for this. First, we feel the DTE module, by using hierarchical assignment rules and not typing network packets, is a lightweight LSM module ideal for use in many situations. Second, as the LSM project is not yet complete, it will benefit from the availability of several modules. This availability will continue to demonstrate the value of LSM to critics, as well as help to keep LSM true to its goal of providing generic support for multiple policies.

We would also like to complete implementation of the extensions listed in Chapter 9.

Addressing the problem of namespaces will soon not be optional. Providing additional methods for specifying file-type resolution will greatly extend the usefulness of DTE. Finally, we would like to implement the protocol shown in Figure 9.4. Following the example of the key server provided by Plan 9 [12], the service would be provided to local servers over a pseudo file system. It would also be loaded as a separate kernel module, for two important reasons. First, it allows us to keep implementations of complicated secret and public key cryptographic functions outside the DTE module. Second, as flaws are found in the implementation, or even the protocol itself, addressing these flaws will not require distribution of a whole new DTE module. This extension would not be required to protect a stand-alone system using DTE, and such a system should therefore not be slowed down or complicated by the extensions, nor be forced to upgrade due to flaws in the extension.

A more dangerous, but promising, endeavor would be to enhance the DTE kernel module such that it can process pcps. A DTE policy could load a set of pcps at boot time, and permit application of modules which do not violate the pcp at run-time. This would be similar to TIS' run-time dynamic policy modification, but with several advantages. The use of arbitrary pcps would enhance both its power and usefulness. The use of modules, rather than policy excerpts, would encourage the application of complete, well thought out modules, rather than the piecemeal application of policy rules. Finally, as a module is considered one entity, its application or refusal would be one atomic action. The question of whether a subset of the submitted rules should be retained if another rule is rejected is moot.

As the number of people using the internet for financial transactions, email, and entertainment purposes continues to increase, computer security is becoming a common topic of

mainstream media. As a result, most people are aware of the inadequate security offered by current operating systems. In this work we presented both a module which significantly enhances the security of the Linux operating system, and tools aimed at simplifying its configuration.

Appendix A

DTE Policy, Modules, and PCP

Listings

A.1 Sample DTE Policies

A.1.1 Base Policy

```
types base_t bin_t boot_t conf_t dev_t disk_t getty_xt init_t \
    lib_t log_t login_et mnt_t oshell_t package_t proc_t root_t sbin_t \
    shell_t tmp_t tty_t user_t varrun_t wdev_t
domains boot_d daemon_d login_d root_d user_d
default_d boot_d

default_et base_t
default_ut base_t

spec_domain boot_d (0) (17 rxld->base_t rxld->bin_t rxlcd->boot_t \
    rxlcd->conf_t rxlcd->dev_t rxld->init_t rxlcd->lib_t rxlcd->log_t \
    rxlcd->mnt_t rxld->oshell_t rxlcd->proc_t rxld->sbin_t rxld->shell_t \
    rxlcd->tmp_t rxlcd->tty_t rxlcd->varrun_t rxlcd->wdev_t) \
    (2 auto->daemon_d auto->login_d) (1 0->0)
spec_domain daemon_d (1 getty_xt) (18 rld->base_t \
    rxld->bin_t rxld->conf_t rxlcd->dev_t rxld->getty_xt rxld->lib_t \
    rxld->log_t rxlcd->mnt_t rxld->oshell_t rxld->package_t \
    rxlcd->proc_t rxld->sbin_t rxld->shell_t rxlcd->tmp_t rxlcd->tty_t \
    rld->user_t rxlcd->varrun_t rxlcd->wdev_t) (1 auto->login_d) (1 0->0)
spec_domain login_d (1 login_et) \
    (16 rld->base_t rxld->bin_t rld->conf_t rxld->lib_t rxlcd->log_t \
    rx->login_et rx->oshell_t rxlcd->proc_t rld->root_t rxld->sbin_t \
```

```

    rx->shell_t rwxlcd->tmp_t rwxlcd->tty_t rld->user_t rwxlcd->varrun_t \
    rwxlcd->wdev_t) (2 exec->root_d exec->user_d) (1 0->0)
spec_domain root_d (1 shell_t) \
(19 rld->base_t rwxlcd->bin_t rwxlcd->conf_t rwxlcd->dev_t rxld->disk_t \
rwxlcd->lib_t rwxlcd->log_t rwxlcd->mnt_t rwxlcd->oshell_t \
rwxlcd->package_t rwxlcd->proc_t rwxlcd->root_t rxld->sbin_t \
rx->shell_t rwxlcd->tmp_t rwxlcd->tty_t rwxlcd->user_t rwxlcd->varrun_t \
rwxlcd->wdev_t) (1 auto->login_d) (1 0->0)
spec_domain user_d (2 shell_t oshell_t) (16 rld->base_t rxld->bin_t \
rxld->conf_t rld->dev_t rxld->lib_t rwxlcd->mnt_t rx->oshell_t \
rxld->package_t rwxlcd->proc_t rxld->sbin_t rx->shell_t rwxlcd->tmp_t \
rwxlcd->tty_t rwxlcd->user_t rwxlcd->varrun_t rwxlcd->wdev_t) \
(1 auto->login_d) (2 14->0 17->0)

```

```

assign -e bin_t /bin
assign -e oshell_t /bin/ash
assign -e oshell_t /bin/bash
assign -e oshell_t /bin/csh
assign -e oshell_t /bin/sh
assign -e oshell_t /bin/tcsh
assign -e shell_t /sbin/sh
assign -e login_et /sbin/login
assign -e boot_t /boot
assign -e dev_t /dev
assign -e tty_t /dev/ttydir
assign -e wdev_t /dev/wdevs
assign -e conf_t /etc
assign -e lib_t /lib
assign -e log_t /log
assign -e disk_t /lost+found
assign -e mnt_t /mnt
assign -e package_t /opt
assign -e proc_t /proc
assign -e root_t /root
assign -e sbin_t /sbin
assign -e init_t /sbin/init
assign -e getty_xt /sbin/mingetty
assign -e tmp_t /tmp
assign -e bin_t /usr/bin
assign -e lib_t /usr/i486-linux-libc5
assign -e lib_t /usr/lib
assign -e lib_t /usr/libexec
assign -e sbin_t /usr/sbin
assign -e tmp_t /usr/tmp
assign -e lib_t /var/lib

```

```

assign -e log_t /var/log
assign -e package_t /var/opt
assign -e varrun_t /var/run
assign -e tmp_t /var/tmp
assign -u bin_t /bin
assign -u boot_t /boot
assign -u dev_t /dev
assign -u tty_t /dev/ttydir
assign -u wdev_t /dev/wdevs
assign -u conf_t /etc
assign -u init_t /etc/rc.d
assign -u user_t /home
assign -u lib_t /lib
assign -u log_t /log
assign -u disk_t /lost+found
assign -u mnt_t /mnt
assign -u package_t /opt
assign -u proc_t /proc
assign -u root_t /root
assign -u sbin_t /sbin
assign -u tmp_t /tmp
assign -u bin_t /usr/bin
assign -u lib_t /usr/i486-linux-libc5
assign -u lib_t /usr/lib
assign -u lib_t /usr/libexec
assign -u sbin_t /usr/sbin
assign -u tmp_t /usr/tmp
assign -u lib_t /var/lib
assign -u log_t /var/log
assign -u package_t /var/opt
assign -u varrun_t /var/run
assign -u tmp_t /var/tmp

```

A.1.2 Password Policy

```

types base_t bin_t boot_t conf_t dev_t disk_t getty_xt init_t \
  lib_t log_t login_et mnt_t oshell_t package_t proc_t root_t sbin_t \
  shell_t tmp_t tty_t user_t varrun_t wdev_t passw_et passw_t shadow_t
domains boot_d daemon_d login_d root_d user_d passw_d
default_d boot_d

default_et base_t
default_ut base_t

```

```

spec_domain boot_d (0) (17 rxld->base_t rxld->bin_t rxwlcd->boot_t \
  rxwlcd->conf_t rxwlcd->dev_t rxld->init_t rxwlcd->lib_t rxwlcd->log_t \
  rxwlcd->mnt_t rxld->oshell_t rxwlcd->proc_t rxld->sbin_t rxld->shell_t \
  rxwlcd->tmp_t rwlcd->tty_t rxwlcd->varrun_t rwlcd->wdev_t) \
  (2 auto->daemon_d auto->login_d) (1 0->0)
spec_domain daemon_d (1 getty_xt) (18 rld->base_t \
  rxld->bin_t rxld->conf_t rxlcd->dev_t rxld->getty_xt rxld->lib_t \
  raxld->log_t rxwlcd->mnt_t rxld->oshell_t rxld->package_t \
  rxwlcd->proc_t rxld->sbin_t rxld->shell_t rxwlcd->tmp_t rwlcd->tty_t \
  rld->user_t rxwlcd->varrun_t rwlcd->wdev_t) (1 auto->login_d) (1 0->0)
spec_domain login_d (1 login_et) \
  (18 rld->base_t rxld->bin_t rld->conf_t rxld->lib_t rxwlcd->log_t \
  rx->login_et rx->oshell_t rxwlcd->proc_t rld->root_t rxld->sbin_t \
  rx->shell_t rxwlcd->tmp_t rwlcd->tty_t rld->user_t rxwlcd->varrun_t \
  rwlcd->wdev_t r->shadow_t r->passw_t) (2 exec->root_d exec->user_d) \
  (1 0->0)
spec_domain root_d (1 shell_t) \
  (21 rld->base_t rxwlcd->bin_t rxwlcd->conf_t rxwlcd->dev_t rxld->disk_t \
  rxwlcd->lib_t rxwlcd->log_t rxwlcd->mnt_t rxwlcd->oshell_t \
  rxwlcd->package_t rxwlcd->proc_t rxwlcd->root_t rxld->sbin_t \
  rx->shell_t rxwlcd->tmp_t rwlcd->tty_t rxwlcd->user_t rxwlcd->varrun_t \
  rwlcd->wdev_t r->passw_et r->passw_t) \
  (2 auto->login_d auto->passw_d) (1 0->0)
spec_domain user_d (2 shell_t oshell_t) (18 rld->base_t rxld->bin_t \
  rxld->conf_t rld->dev_t rxld->lib_t rxwlcd->mnt_t rx->oshell_t \
  rxld->package_t rxwlcd->proc_t rxld->sbin_t rx->shell_t rxwlcd->tmp_t \
  rwlcd->tty_t rxwlcd->user_t rxwlcd->varrun_t rwlcd->wdev_t r->passw_et \
  r->passw_t) \
  (2 auto->login_d auto->passw_d) (2 14->0 17->0)
spec_domain passw_d (1 passw_et) (7 rld->base_t rld->user_t rx->passw_et \
  rwlcd->passw_t rwlcd->shadow_t rld->lib_t rwlcd->log_t) \
  (0) (14->0 17->0)

```

```

assign -e bin_t /bin
assign -e oshell_t /bin/ash
assign -e oshell_t /bin/bash
assign -e oshell_t /bin/csh
assign -e oshell_t /bin/sh
assign -e oshell_t /bin/tcsh
assign -e shell_t /sbin/sh
assign -e login_et /sbin/login
assign -e boot_t /boot
assign -e dev_t /dev
assign -e tty_t /dev/ttydir

```



```
assign -e wdev_t /dev/wdevs
assign -e conf_t /etc
assign -e lib_t /lib
assign -e log_t /log
assign -e disk_t /lost+found
assign -e mnt_t /mnt
assign -e package_t /opt
assign -e proc_t /proc
assign -e root_t /root
assign -e sbin_t /sbin
assign -e init_t /sbin/init
assign -e getty_xt /sbin/mingetty
assign -e tmp_t /tmp
assign -e bin_t /usr/bin
assign -e lib_t /usr/i486-linux-libc5
assign -e lib_t /usr/lib
assign -e lib_t /usr/libexec
assign -e sbin_t /usr/sbin
assign -e tmp_t /usr/tmp
assign -e lib_t /var/lib
assign -e log_t /var/log
assign -e package_t /var/opt
assign -e varrun_t /var/run
assign -e tmp_t /var/tmp
assign -u bin_t /bin
assign -u boot_t /boot
assign -u dev_t /dev
assign -u tty_t /dev/ttydir
assign -u wdev_t /dev/wdevs
assign -u conf_t /etc
assign -u init_t /etc/rc.d
assign -u user_t /home
assign -u lib_t /lib
assign -u log_t /log
assign -u disk_t /lost+found
assign -u mnt_t /mnt
assign -u package_t /opt
assign -u proc_t /proc
assign -u root_t /root
assign -u sbin_t /sbin
assign -u tmp_t /tmp
assign -u bin_t /usr/bin
assign -u lib_t /usr/i486-linux-libc5
assign -u lib_t /usr/lib
assign -u lib_t /usr/libexec
```

```
assign -u sbin_t /usr/sbin
assign -u tmp_t /usr/tmp
assign -u lib_t /var/lib
assign -u log_t /var/log
assign -u package_t /var/opt
assign -u varrun_t /var/run
assign -u tmp_t /var/tmp
assign -e shadow_t /pwd/shadow
assign -e passwd_t /pwd/passwd /pwd/passwd.tmp /pwd/.pwd.lock
assign -e passwd_et /sbin/passwd
```

A.2 Sample DTE Modules

A.2.1 Base Module

Module System.Base

```
type base_t

    DEFAULT_RTYPE

    absolute access all rld

    absolute access boot_d rxld

end

type bin_t

    upath /bin

    upath /usr/bin

    upath /usr/local/bin

    absolute access all rxld

    access Admin.services.+ rwxlcd

end

type boot_t
```

```
    rpath /boot

    absolute access all none

    access boot_d rwxlcd

    access Admin.kernel rwlcd

end

type conf_t

    access boot_d rxlcd

    access Admin.config rwxlcd

    absolute access all rxld

    rpath /etc

end

type dev_t

    rpath /dev

    absolute access getty_d rxld

    access boot_d rwxlcd

    access login_d rwlcd

    access Admin.config rwxlcd

    absolute access all rld

end

type disk_t

    absolute access all none

    access Admin.services.+ rwxlcd

    rpath /lost+found
```

```
end

type getty_xt
    epath /sbin/mingetty
    access getty_d rx
end

type init_t
    absolute access boot_d rxld
    epath /sbin/init
    upath /etc/rc.d
end

type lib_t
    absolute access all rxld
    access boot_d rwxlcd
    access Admin.services.+ rwxlcd
    rpath /lib /usr/lib /usr/local/lib /usr/libexec /var/lib
    rpath /usr/i486-linux-libc5
end

type log_t
    access boot_d rwxlcd
    access getty_d raxld
    access login_d rwxlcd
    access root_d rwxlcd
    access Admin.+ rld
```

```
    absolute access all none

    rpath /log

    rpath /var/log

end

type login_et

    epath /bin/login

    epath /bin/su

    absolute access login_d rx

    absolute access all none

    access Admin.services.+ rwxlcd

end

type oshell_t

    epath /bin/ash

    epath /bin/csh

    epath /bin/tcsh

    access all rx

    access Admin.* rwxlcd

end

type proc_t

    rpath /proc

    access all rwxlcda

end

type root_t
```

```
    rpath /root

    absolute access all none

    access root_d rwxlcd

end

type sbin_t

    rpath /sbin /usr/sbin /usr/local/sbin

    absolute access all rxld

    access Admin.sbin rwxlcd

end

type shell_t

    epath /bin/bash

    epath /bin/sh

    access all rx

end

type tmp_t

    access all rwxlcd

    rpath /tmp /usr/tmp /var/tmp

end

type tty_t

    access all rwlcd

    rpath /dev/ttydir

end

type user_t
```

```
    access getty_d rld

    access login_d rld

    access Admin.* rld

    access user_d rwxlcd

    upath /home

end

type varrun_t

    access all rwxlcd

    rpath /var/run

end

type wdev_t

    rpath /dev/wdevs

    absolute access all rwlcd

end

domain System.boot_d

    DEFAULT_DOMAIN

    domain out getty_d auto

    signal out all 0

end

domain Service.getty_d

    entries getty_xt

    domain out login_d auto
```

```
domain in boot_d auto

domain in Admin.services.+ exec

signal out all 0

end

domain Service.login_d

entries login_et

domain in Admin.services.+ exec

domain in login_d auto

domain out Users.* exec

signal out all 0

end

domain Users.root_d

entries shell_t

domain in login_d auto

domain out all none

signal out all 0

end

domain Users.user_d

entries shell_t oshell_t

domain in login_d auto

absolute domain out Users.* none

signal out all 14,17

end
```


end

A.2.2 Password Module

Module Service.password

domain passw_d

entries passw_et

domain in Users.* auto

absolute domain out all none

type all none

type user_t rld

signal out all 14,17

signal in all none

end

type passw_et

absolute access all r

access passw_d rx

epath /bin/passwd /usr/bin/passwd

end

type passw_t override

absolute access all rld

access passw_d rwlcd

epath /etc/passwd

```
        epath /etc/passwd.tmp /etc/.pwd.lock

    end

    type shadow_t

        absolute access all none

        access passwd rwlcd

        access login_d rld

        epath /etc/shadow /etc/shadow.tmp

    end

end
```

A.2.3 Ftp Module

Module Service.ftp

```
    domain ftpd_d

        entries ftpd_et

        domain in all none

        domain out all none

        domain in boot_d auto

        domain in Admin.services.+ exec

        absolute type all none

        signal out boot_d 14,17

        signal out Admin.services.+ 14,17

    end
```

```
type ftp_t
    aaccess all none
    absolute access ftpd_d rld
    rpath /home/ftp
end

type ftpd_et
    access all r
    absolute access ftpd_d rx
    epath /usr/sbin/in.ftpd
end

type ftpd_xt
    access all none
    absolute access ftpd_d rxld
    access root_d rwclld
    rpath /home/ftp/bin
    assert blp protect
end

type ftpd_wt
    access all none
    absolute access ftpd_d rwclld
```

```
        rpath /home/ftp/incoming
    end
end
```

A.2.4 Syslog Module

Module Service.syslog

```
domain syslog_d

    entries syslog_xt # /usr/sbin/secure_syslog_d

    domain in all none

    domain in boot_d auto

    domain out all none

    type urandom_t r

    type dev_t r # in case no urandom type

    signal in all none

    signal out boot_d 14,17

end

type syslog_xt

    access all none

    access boot_d r

    access syslog_d rx

    epath /usr/sbin/secure_syslog_d

end
```

```

type syslog_in_t

    access all a

    access syslog_d r

    epath /var/log/secure_log_in

end

type syslog_out_t

    access all none

    access root_d r

    access syslog_d ra

    epath /var/log/secure_log_out

end

end

```

A.3 Excerpts of BLP PCP

Following are excerpts of the code for the BLP pcp.

A.3.1 Finding Paths

The following code finds paths within the \leq graph.

```

00: sub path_exists_orig {
01:   my ($search_leq) = shift @_;
02:   my ($a, $b) = @_;
03:   my ($t1, $t2);
04:   my (%leq2);
05:
06:   foreach $t1 (keys (%$search_leq)) {
07:     next unless $t1 eq $a;
08:     return "true" if leq_contains($$search_leq{$a}, $b);

```

```

09:
10:     while (defined($$search_leq{$t1}) and $$search_leq{$t1} ne '') {
11:         ($t2, $$search_leq{$t1}) = split ':', $$search_leq{$t1}, 2;
12:         %leq2 = %$search_leq;
13:         if (path_exists_orig(\%leq2, $t2, $b)) {
14:             return "true";
15:         }
16:     }
17: }
18: return '';
19: }

```

A.3.2 Relation Calculation

This is the code to compute BLP relation resulting from a DTE policy.

```

00: sub calculate_blp {
01:     my (%leq);
02:     my ($dom, $domain, %hlist, $type1, $type2, $value, $value2);
03:
04:     foreach $dom (keys (%main::domains)) {
05:         $domain = $main::domains{$dom};
06:         next unless exists $domain->{"realta"};
07:         %hlist = %{$domain->{"realta"}};
08:         foreach $type1 (keys (%hlist)) {
09:             $value = $hlist{$type1}->[0];
10:             next unless index($value, "r") != -1;
11:             foreach $type2 (keys (%hlist)) {
12:                 $value2 = $hlist{$type2}->[0];
13:                 next unless has_modify_acc($domain, $type2);
14:                 append_leq(\%leq, $type1, $type2);
15:             }
16:         }
17:     }
18:     return %leq;
19: }

```

A.3.3 Pre-Apply

This is the BLP pre_apply function.

```

00: sub pre_apply {

```

```

01:   my ($self) = @_;
02:   my ($type, $domain);
03:   %baset = %main::types;
04:   %leq = calculate_blp();
05:   print "Pre-apply ";
06:   print_blp(%leq);
07: }

```

A.3.4 Post-Apply

This is the BLP post_apply function.

```

00: sub post_apply {
01:   my ($self) = @_;
02:   my ($a, $b, $path_exists);
03:
04:   %post_leq = calculate_blp();
05:   foreach $a (keys (%post_leq)) {
06:     next unless defined $baset{$a};
07:     foreach $b (split ':', $post_leq{$a}) {
08:       next unless defined $baset{$b};
09:       next if $a eq $b;
10:       %search_leq = %leq;
11:       $path_exists = path_exists_orig(\%search_leq, $a, $b);
12:       if (not $path_exists) {
13:         print "BLP VIOLATION: $a newly <= $b.\n";
14:       }
15:     }
16:   }
17: }

```

A.4 Excerpts of MBLP PCP

Following are excerpts of the code for the MBLP pcp.

A.4.1 Relation Calculation

```

00: sub calculate_mblp {
01:   my (%leq);
02:   my ($dom, $domain, %hlist, $type1, $type2, $value, $value2);

```

```

03:
04:   foreach $dom (keys (%main::domains)) {
05:     $domain = $main::domains{$dom};
06:     next unless exists $domain->{"realta"};
07:     next if is_ignore_domain($dom);
08:     %hlist = %{$domain->{"realta"}};
09:     foreach $type1 (keys (%hlist)) {
10:       $value = $hlist{$type1}->[0];
11:       next unless index($value, "r") != -1;
12:       next if is_ignore_type($type1);
13:       next if is_secret_exception($type1, $dom);
14: # Now, since dom is not a secrecy exception:
15:       if (is_secret_type($type1)) {
16:         print "MOD_BLP: $dom should NOT read type $type1.\n";
17:       }
18:       foreach $type2 (keys (%hlist)) {
19:         $value2 = $hlist{$type2}->[0];
20:         next unless has_modify_acc($domain, $type2);
21:         next if is_ignore_type($type2);
22:         next if is_protect_exception($type2, $dom);
23: # Now, since dom is not a protection exception:
24:         if (is_protected_type($type2)) {
25:           print "MOD_BLP: $dom should NOT write type $type2.\n";
26:         }
27:         append_leq(\%leq, $type1, $type2);
28:       }
29:     }
30:   }
31:
32:   return %leq;
33: }

```

A.4.2 Pre-Apply

```

00: sub pre_apply {
01:   my ($self) = @_;
02:   my ($type, $domain);
03:   %baset = %main::types;
04:   setup_asserts();
05:   %leq = calculate_mblp();
06: }

```


A.4.3 Post-Apply

```
00: sub post_apply {
01:   my ($self) = @_;
02:   my ($a, $b, $path_exists);
03:
04:   %post_leq = calculate_mblp();
05: # for each a,b, post_leq{$a} =b, and a,b both in base_types, make sure
06: # leq{$a} = $b;
07:   foreach $a (keys (%post_leq)) {
08:     next unless defined $baset{$a};
09:     foreach $b (split ':', $post_leq{$a}) {
10:       next unless defined $baset{$b};
11:       next if $a eq $b;
12:       %search_leq = %leq;
13:       $path_exists = path_exists_orig(\%search_leq, $a, $b);
14:       if (not $path_exists) {
15:         print "MBLP VIOLATION: $a newly <= $b.\n";
16:       }
17:     }
18:   }
19: }
```

Appendix B

LMBench Results

LMBench [36] was used to measure the performance of the stock 2.5.6 linux kernel, as well as the LSM 2.5.6 kernel. The LSM kernel was tested using the dummy module, the capabilities module, the DTE module, and the capabilities module stacked on top of the DTE module. The results are listed here. For the sake of brevity and readability, we report only the mean and deviation of 10 runs.

B.1 Stock Kernel

Host	OS Description	Mhz
skull-spl	Linux 2.5.6 i686-pc-linux-gnu	398

Processor, Processes - times in microseconds - smaller is better

	Mhz	null call	null I/O	stat	open clos	selct TCP	sig inst	sig hndl	fork proc	exec proc
mean		0.90	1.61	9.15	11.9	62.5	2.23	6.99	475	1846
sigma		0.01	0.03	.09	0.2	2.4	.02	.03	5	16
95% CI		.008	.023	.068	.151	1.81	.015	.023	3.77	12

Context switching - times in microseconds - smaller is better

	2p/0K	2p/16K	2p/64K	8p/16K	8p/64K	16p/16K	16p/64K
ctxsw	ctxsw	ctxsw	ctxsw	ctxsw	ctxsw	ctxsw	ctxsw

```

-----
mean          2.338  18.8  55.7  19.9  132.0  26.5  205.4
sigma         0.140   0.3   5.6   0.8   9      3.0   2.0
95% CI        .106   .226  4.22  .603  6.79   2.26  1.51

```

Local Communication latencies in microseconds - smaller is better

```

-----
                2p/OK Pipe AF      UDP  RPC/  TCP  RPC/ TCP
                ctxsw      UNIX      UDP      TCP conn
-----
mean          2.338  13.2 26.0  47.8 112.7  73.3 158.3  255
sigma         0.140   0.4 0.3  0.60  0.5   1.1  1.9   1
95% CI        .106  .302 .226  .452  .377  .829  1.43  .754

```

File & VM system latencies in microseconds - smaller is better

```

-----
                OK File      10K File      Mmap  Prot  Page
                Create Delete Create Delete Latency Fault Fault
-----
mean          123.6  15.6 259.0  38.9  1301.5 1.702 5.70000
sigma         0.2    0.1  0.2   0.1   23.9 0.022 0.45826
95% CI        .151  .075 .151  .075   18.0 .017  .346

```

Local Communication bandwidths in MB/s - bigger is better

```

-----
                Pipe AF  TCP  File  Mmap  Bcopy  Bcopy  Mem  Mem
                UNIX      reread reread (libc) (hand) read write
-----
mean          177  115  122  149.3  277.8  135.7  142.4  277  170.11
sigma         11   5   47   0.2   0.0   0.7   1.1   0   0.03
95% CI        8.29 3.77 35.4  .151  .000  .528  .829 .000  .023

```

B.2 LSM Kernel Using Dummy Module

Processor, Processes - times in microseconds - smaller is better

```

-----
                Mhz null null      open selct sig  sig  fork exec
                call  I/O stat clos TCP  inst hndl  proc proc
-----
mean          0.859 1.61 9.49 12.0  64.8 2.20 7.10  474 1870
sigma         0.003 0.02 0.10 0.1   3.0 0.06 0.01   3   16
95% CI        .002 .015 .075 .075  2.26 .045 .008 2.26 12.1

```

Context switching - times in microseconds - smaller is better

	2p/0K ctxsw	2p/16K ctxsw	2p/64K ctxsw	8p/16K ctxsw	8p/64K ctxsw	16p/16K ctxsw	16p/64K ctxsw
mean	2.558	19.2	54.6	21.4	150.2	27.4	202.6
sigma	0.172	0.1	1.3	3.9	12.1	2.8	2.6
95% CI	.130	.075	.980	2.94	9.12	2.11	1.96

Local Communication latencies in microseconds - smaller is better

	2p/0K ctxsw	Pipe AF UNIX	UDP	RPC/ UDP	TCP	RPC/ TCP	TCP conn	
mean	2.558	13.4	25.6	45.2	114.0	76.5	157.3	257
sigma	0.172	0.4	0.4	0.7	0.5	0.6	7.8	2
95% CI	.130	.302	.302	.528	.377	.452	5.88	1.51

File & VM system latencies in microseconds - smaller is better

	OK Create	File Delete	10K Create	File Delete	Mmap Latency	Prot Fault	Page Fault
mean	126.3	17.4	260.7	41.8	1295.3	1.689	5.40
sigma	0.3	0.1	0.4	0.2	20	0.024	0.49
95% CI	.226	.075	.302	.151	15.1	.018	.369

Local Communication bandwidths in MB/s - bigger is better

	Pipe UNIX	AF	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
mean	187	118	80.8	149.4	277.8	135.6	142.3	277	170.2
sigma	3	3	3.1	0.1	0.04	0.7	1.1	0	0.1
95% CI	2.26	2.26	2.34	.075	.030	.528	.829	.000	.075

B.3 LSM Kernel Using Capabilities Module

Processor, Processes - times in microseconds - smaller is better

Mhz	null	null	open	selct	sig	sig	fork	exec
-----	------	------	------	-------	-----	-----	------	------

	call	I/O	stat	clos	TCP	inst	hndl	proc	proc
mean	0.859	1.62	9.52	12.1	64.5	2.18	7.10	475	1870
sigma	0.003	0.03	0.13	0.2	2.3	0.00	0.01	3	20
95% CI	.002	.023	.098	.151	1.73	.000	.008	2.26	15.1

Context switching - times in microseconds - smaller is better

	2p/OK ctxsw	2p/16K ctxsw	2p/64K ctxsw	8p/16K ctxsw	8p/64K ctxsw	16p/16K ctxsw	16p/64K ctxsw
mean	2.55	19.3	56.2	21.0	143.5	30.2	205.5
sigma	0.11	0.2	5.6	1.7	7.5	4.9	2.3
95% CI	.081	.151	4.22	1.28	5.66	3.69	1.73

Local Communication latencies in microseconds - smaller is better

	2p/OK ctxsw	Pipe ctxsw	AF UNIX	UDP	RPC/ UDP	TCP	RPC/ TCP	TCP conn
mean	2.55	13.7	25.8	45.8	113.8	76.2	161.8	256
sigma	0.11	0.3	0.3	0.6	1.2	1.1	1.6	1.4
95% CI	.083	.226	.226	.452	.905	.829	1.21	1.06

File & VM system latencies in microseconds - smaller is better

	OK File Create	File Delete	10K File Create	File Delete	Mmap Latency	Prot Fault	Page Fault
mean	126.0	17.0	260.6	41.9	1298	1.730	5.60
sigma	0.2	0.1	0.3	0.1	18	0.031	0.49
95% CI	.151	.075	.226	.075	13.6	.023	.369

Local Communication bandwidths in MB/s - bigger is better

	Pipe UNIX	AF	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
mean	181	117	80.8	148.6	277.8	136.4	143.4	277	170.1
sigma	9	4	7.3	1.9	0.05	0.2	0.3	0	0.05
95% CI	6.79	3.02	5.50	1.43	.038	.151	.226	.000	.038

B.4 LSM Kernel Using DTE Module

Processor, Processes - times in microseconds - smaller is better

	Mhz	null call	null I/O	stat	open clos	selct TCP	sig inst	sig hdl	fork proc	exec proc
mean		0.86	1.62	12.2	15.4	67.0	2.18	7.20	480	1885
sigma		0	0.01	0.1	0.1	3.6	0	0.01	4	18
95% CI		.000	.008	.075	.075	2.71	.000	.008	3.02	13.6

Context switching - times in microseconds - smaller is better

	2p/OK ctxsw	2p/16K ctxsw	2p/64K ctxsw	8p/16K ctxsw	8p/64K ctxsw	16p/16K ctxsw	16p/64K ctxsw
mean	2.424	19.5	54.2	20.7	145.0	28.9	206.6
sigma	0.136	0.1	0.4	0.9	15.8	4.3	1.5
95% CI	.103	.075	.302	.679	12.0	3.24	1.13

Local Communication latencies in microseconds - smaller is better

	2p/OK ctxsw	Pipe AF UNIX	UDP	RPC/ UDP	TCP	RPC/ TCP conn	TCP
mean	2.424	13.7	25.8	45.0	113.3	77.7	162.6
sigma	0.136	0.2	0.4	0.6	0.9	1.8	1.5
95% CI	.103	.151	.302	.452	.679	1.36	1.13

File & VM system latencies in microseconds - smaller is better

	OK Create	File Delete	10K Create	File Delete	Mmap Latency	Prot Fault	Page Fault
mean	131.6	19.4	268.3	44.8	1295	1.640	5.80
sigma	0.3	0.1	0.3	0.1	24	0.033	0.40
95% CI	.226	.075	.226	.075	18.1	.025	.302

Local Communication bandwidths in MB/s - bigger is better

	Pipe UNIX	AF	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
mean	182	118	88.8	149.3	277.8	136.5	143.7	277	170.16
sigma	2	2	28.3	0.1	0.03	0.11	0.3	0	0.05

95% CI 1.51 1.51 21.3 .075 .023 .083 .226 .000 .038

B.5 LSM Kernel Using DTE and Capabilities Module

Processor, Processes - times in microseconds - smaller is better

```
-----
                Mhz null null      open selct sig  sig  fork exec
                call  I/O stat clos TCP  inst hndl  proc proc
-----
mean                0.859 1.62 12.2 15.5 68.0 2.183 7.20 479 1903
sigma                0.003 0.01 0.1 0.3 4.7 .004 0.01 4 37
95% CI                .002 .008 .075 .226 3.54 .003 .008 3.02 27.9
```

Context switching - times in microseconds - smaller is better

```
-----
                2p/OK 2p/16K 2p/64K 8p/16K 8p/64K 16p/16K 16p/64K
                ctxsw ctxsw  ctxsw ctxsw  ctxsw  ctxsw  ctxsw
-----
mean                2.531 19.4 54.4 20.7 147.5 29.3 203.3
sigma                0.167 0.28 0.7 0.7 20.7 3.3 5.3
95% CI                .126 .211 .528 .528 15.6 2.49 4.00
```

Local Communication latencies in microseconds - smaller is better

```
-----
                2p/OK Pipe AF      UDP RPC/  TCP  RPC/ TCP
                ctxsw      UNIX      UDP  UDP  TCP  TCP conn
-----
mean                2.531 13.8 25.8 45.0 113.4 76.7 162.8 260
sigma                0.167 0.4 0.3 0.4 0.5 1.1 1.7 2
95% CI                .126 .302 .226 .302 .377 .829 1.28 1.51
```

File & VM system latencies in microseconds - smaller is better

```
-----
                OK File      10K File      Mmap  Prot  Page
                Create Delete Create Delete Latency Fault Fault
-----
mean                131.6 19.3 268.3 44.9 1296.3 1.613 5.40
sigma                0.3 0.1 0.4 0.1 22 0.019 0.49
95% CI                .226 .075 .302 .075 16.6 .014 .369
```

Local Communication bandwidths in MB/s - bigger is better

```
-----
```

	Pipe	AF	TCP	File	Mmap	Bcopy	Bcopy	Mem	Mem
	UNIX			reread	reread	(libc)	(hand)	read	write
	-----	-----	-----	-----	-----	-----	-----	-----	-----
mean	183	118	103	149.3	277.79	136.6	143.7	277	170.1
sigma	5	2	40	0.1	0.03	0.1	0.1	0	0.049
95% CI	3.77	1.51	30.2	.075	.023	.075	.075	.000	.037

Bibliography

- [1] NATIONAL SECURITY AGENCY. Evaluated platforms: Trusted xenix. <http://www.radium.ncsc.mil/tpep/epl/entries/CSC-EPL-92-001-A.html>.
- [2] LEE BADGER, DANIEL F. STERNE, DAVID L. SHERMAN, KENNETH M. WALKER, AND SHEILA A. HAGHIGHAT. A domain and type enforcement unix prototype. *Proceedings of the Fifth USENIX UNIX Security Symposium*, June 1995.
- [3] LEE BADGER, DANIEL F. STERNE, DAVID L. SHERMAN, KENNETH M. WALKER, AND SHEILA A. HAGHIGHAT. Practical domain and type enforcement for unix. *Proceedings of the IEEE Symposium on Security and Privacy*, pages 66–77, May 1995.
- [4] D.E. BELL AND L.J. LA PADULA. Security computer systems : Unified exposition and multics interpretation. Technical report, Hanscom AFB. Bedford. MA. Rep. ESD-TR-75-306., March 1976.
- [5] K. J. BIBA. Integrity considerations for secure computer systems. *USAF Electronic Systems Division*, 1977.
- [6] W.E. BOEBERT AND R.Y. KAIN. A practical alternative to hierarchical integrity policies. *Proceedings of the National Computer Security Conference*, 8:18, 1985.
- [7] DANIEL P. BOVET AND MARCO CESATI. *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, January 2001.
- [8] TOM CHRISTIANSEN. Interface zen, November 1999. <http://slashdot.org/features/99/11/30/0954216.shtml>.
- [9] DAVID D. CLARK AND DAVID R. WILSON. A comparison of commercial and military computer security policies. *Proceedings of the IEEE Symposium on Security and Privacy*, pages 184–194, 1987.
- [10] SECURE COMPUTING. Type enforcement technology for access gateways and vpns. <http://www.secure-computing.com>.
- [11] CRISPIN COWAN, April 2001. <http://mail.wirex.com/pipermail/linux-security-module/2001-April:/0005.html>.
- [12] RUSS COX, ERIC GROSSE, ROB PIKE, DAVE PRESOTTO, AND SEAN QUINLAN. Security in plan 9. *Proceedings of the Usenix Security Symposium*, 11, August 2002.

- [13] CHRIS DALTON AND TSE HUONG CHOO. An operating system approach to securing e-services. *Communications of the ACM*, 44, 2001.
- [14] JACK B. DENNIS AND EARL C. VAN HORN. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3):143–155, March 1966.
- [15] SOLAR DESIGNER. <http://www.openwall.com/>.
- [16] NIGEL EDWARDS, JOUBERT BERGER, AND TSE-HUONG CHOO. A secure linux platform. *Proceedings of the Annual Linux Showcase and Conference*, 5, November 2001.
- [17] ION BADULESCU EREZ ZADOK AND ALEX SHENDER. Extending file systems using stackable templates. *Proceedings of the 1999 USENIX Annual Technical Conference*, 1999.
- [18] CRISPIN COWAN ET AL. <http://www.immunix.org/>.
- [19] CRISPIN COWAN ET AL. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *Usenix Security Conference*, 7, January 1998.
- [20] SECURITY FOCUS. Enterprise security and threat management systems. <http://www.securityfocus.com>.
- [21] RON FORRESTER AND BRIAN ROBISON. <http://www.tripwire.org/>.
- [22] TIMOTHY FRASER. Lomac: Mac you can live with. *Usenix Annual Technical Conference*, 2001.
- [23] TIMOTHY FRASER AND LEE BADGER. Ensuring continuity during dynamic security policy reconfiguration in dte. *Proceedings of the IEEE Symposium on Security and Privacy*, 1998.
- [24] VIRGIL D. GLIGOR, SERBAN I. GAVRILA, AND DAVID FERRAILOLO. On the formal definition of separation-of-duty policies and their composition. *Proceedings of the IEEE Symposium on Security and Privacy*, May 1998.
- [25] POSIX SECURITY WORKING GROUP. Posix system api amendment 1003.1e: Protection, audit and control interfaces (*withdrawn*), oct 1997.
- [26] ANDREAS GRUNBACHER. <http://acl.bestbits.at>.
- [27] SERGE HALLYN AND PHIL KEARNS. Domain and type enforcement for linux. *Proceedings of the Atlanta Linux Showcase*, 4, October 2000.
- [28] INTEL. *Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. 1997.
- [29] MICHAEL JOHNSON. A tour of the linux vfs, 1996. <http://www.linuxdoc.org/LDP/khg/HyperNews/get/fs/vfstour.html>.
- [30] ALEXANDER KJELDAAS. Linux capability faq v0.1, 1998. <http://www.uwsg.indiana.edu/hypermil/linux/kernel/9808.1/0178.html>.

- [31] GREG KROAH-HARTMAN. How the pci hot plug driver filesystem works. *Linux Journal*, 97, May 2001.
- [32] CHUCK LEVER. Linux kernel hash table behavior: Analysis and improvements. *Proceedings of the Atlanta Linux Showcase*, 4, October 2000.
- [33] PETER LOSCOCCO AND STEPHEN SMALLEY. Integrating flexible support for security policies into the linux operating system. *Usenix Technical Conference*, June 2001.
- [34] PETER A. LOSCOCCO AND STEPHEN D. SMALLEY. Meeting critical security objectives with security-enhanced linux. *Proceedings of the Ottawa Linux Symposium*, July 2002.
- [35] DURWARD McDONNELL, DAVID SAMES, GREGG TALLY, AND ROBB LYDA. Security for distributed object-oriented systems. *DARPA Information Survivability Conference and Exposition*, June 2001.
- [36] LARRY McVOY AND CARL STAELIN. lmbench: Portable tools for performance analysis. *Usenix Technical Conference*, 1996.
- [37] R. O'BRIEN AND C. ROGERS. Developing applications on lock. *Proceedings of the National Computer Security Conference*, 14:147–156, October 1991.
- [38] AMON OTT. The role compatibility security model. *Nordic Workshop on Secure IT Systems*, November 2002. <http://www.rsbac.org/rc-nordsec2002>.
- [39] PRZEMYSŁAW PARDYAK AND BRIAN N. BERSHAD. Dynamic binding for an extensible system. *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 201–212, 1996.
- [40] ROB PIKE, DAVE PRESOTTO, KEN THOMPSON, HOWARD TRICKEY, AND PHIL WINTERBOTTOM. The use of name spaces in plan 9. *Operating Systems Review*, 27(2):72–76, April 1993.
- [41] SAIC. Access control methods. <http://research-cistw.saic.com/cace/>.
- [42] BRUCE SCHNEIER. *Applied Cryptography*. John Wiley & Sons, Inc, New York, NY, 2 edition, 1996. Chapter 4.2: Subliminal Channels.
- [43] BRUCE SCHNEIER AND JOHN KELSEY. Cryptographic support for secure logs on untrusted machines. *Proceedings of the Seventh USENIX Security Symposium*, pages 53–62, January 1998.
- [44] STEPHEN SMALLEY. Configuring the selinux policy, June 2002. <http://www.nsa.gov/selinux/Documentation/WHEREISIT/>.
- [45] RAY SPENCER, STEPHEN SMALLEY, PETER LOSCOCCO, MIKE HIBLER, DAVE ANDERSEN, AND JAY LEPREAU. The flask security architecture: System support for diverse security policies. *Usenix Security Symposium*, 9, 1999.
- [46] NAI STEPHEN SMALLEY. <http://www.cs.utah.edu/flux/fluke/html/linux.html>.

- [47] RITA C. SUMMMERS. *Secure Computing: Threats and Safeguards*. McGraw-Hill, NY, 1997.
- [48] ANDREW S. TANENBAUM. *Modern Operating Systems*. Prentice Hall, Englewood Cliffs, NJ, 1992.
- [49] CMU COMPUTER EMERGENCY RESPONSE TEAM. Cert advisory ca-2000-13: Two input validation problems in ftpd. <http://www.cert.org/advisories/CA-2000-13.html>.
- [50] LSM DEVELOPMENT TEAM. Linux security modules, 2001. <http://lsm.immunix.org/>.
- [51] LINUS TORVALDS. Call for a general module extension for access control in linux, 2001. <http://mail.wirex.com/pipermail/linux-security-module/2001-April/000005.html>.
- [52] ALEXANDER VIRO. Per-process namespaces for linux. <http://lwn.net/2001/0301/a/namespaces.php3>.
- [53] KENNETH M. WALKER, DANIEL F. STERNE, M. LEE BADGER, MICHAEL J. PETKAC, DAVID L. SHERMANN, AND KAREN A. OOSTENDORP. Confining root programs with domain and type enforcement(dte). *USENIX UNIX Security Symposium*, 6, 1996.
- [54] CHRIS WRIGHT, CRISPIN COWAN, STEPHEN SMALLEY, JAMES MORRIS, AND GREG KROAH-HARTMAN. Linux security modules: General security support for the linux kernel. *Proceedings of the Eleventh Usenix Security Symposium*, 11, August 2002.
- [55] THOMAS WU. The secure remote password protocol. *Proceedings of Internet Society Network and Distributed System Security Symposium*, pages 97–111, March 1998.
- [56] HUAGANG XIE AND PHILIPPE BIONDI. <http://www.lids.org>.
- [57] EREZ ZADOK AND JASON NIEH. Fist: A language for stackable file systems. *Proceedings of the USENIX Technical Conference*, June 2000.

VITA

Serge Edward Hallyn

Serge Hallyn was born in Brugge, Belgium. He graduated in 1992 from Downers Grove North High School in Downers Grove, Illinois. He obtained a BS in computer science and physics from Hope College, MI, in 1996. In 1998, he obtained his MS in computer science from the College of William and Mary, in Williamsburg, Virginia, where he continued to pursue his Ph.D.