AFRL-IF-RS-TR-2002-230
Final Technical Report
September 2002

EXTENSIBLE OPERATING SYSTEM SECURITY

Trusted Information Systems Inc.

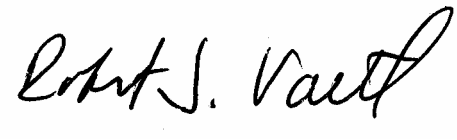
Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. F207

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

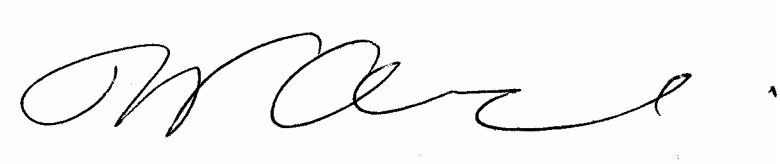
AFRL-IF-RS-TR-2002-230 has been reviewed and is approved for publication.

APPROVED:



ROBERT J. VAETH
Project Engineer

FOR THE DIRECTOR:



WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 2002	**Final** **Aug 97 – Sep 2000**

4. TITLE AND SUBTITLE

EXTENSIBLE OPERATING SYSTEM SECURITY

5. FUNDING NUMBERS

C - F30602-97-C-0258
PE - 62301E
PR - F207
TA - 71
WU - 01

6. AUTHOR(S)

Dennis Hollingworth, Timothy Redmond and Robert Rice

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Trusted Information Systems, Inc.
3415 S. Sepulveda Blvd., Suite 700
Los Angeles CA 90034

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFGB
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2002-230

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Robert J. Vaeth/IFG/(315) 330-2182/Robert.Vaeth@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

The EXOS project investigated the practical application of security to extensible operating systems. The project leaders investigated how security policies should be represented and supported in extensible systems, concluding that classical policy models are intuitive, capture widely accepted policy requirements, and provide a concrete foundation for systems evaluation efforts. The investigators introduced support for several well known, important user-level security policy models. In addition, the investigators introduced kernel-oriented security by supporting the separation of kernel code into segments and enforcing a domain/type policy on threads as they execute or otherwise access system segments. Finally, they devised a domain/type policy extensibility strategy that is conservative, preserving prior policy after allowed extensions.

14. SUBJECT TERMS

Operating Systems, Extensibility, Security, Policy

15. NUMBER OF PAGES

35

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Preface

Network Associates is pleased to present this report as CDRL A008 of Contract F30602-97-C-0258 for the Extensible Operating System Security (EXOS) project.

The EXOS project investigated the practical application of security to extensible operating systems, focusing on the University of Washington's SPIN extensible operating system. SPIN supports dynamic kernel extension by untrusted user code that downloads code and data directly into the kernel. Because of this high degree of kernel dynamism, we conclude that kernel-oriented security policy support is required in such systems in addition to traditional user-level security policy support, to prevent kernel extensions from subverting kernel functionality. In order to establish that our approach to extensible OS security is practical, we augmented the existing security architecture of SPIN with access-controlled software segments while preserving the SPIN model of separation between policy mediation and enforcement. We investigated how security policies should be represented and supported in extensible systems, concluding that classical policy models are intuitive, capture widely accepted policy requirements, and provide a concrete foundation for systems evaluation efforts. We introduced support for several well-known, important user-level security policy models. In addition, we introduced kernel-oriented security by supporting the separation of kernel code into segments and enforcing a domain/type policy on threads as they execute or otherwise access system segments. Finally, we devised a domain/type policy extensibility strategy that is *conservative*, preserving prior policy after allowed extensions.

This report has six major sections. The introduction briefly explores basic issues and addresses what we mean when we use the term *extensible operating system* and how this concept relates to the traditional notion of the kernel as part of the Trusted Computing Base. The second section explores and motivates our ideas surrounding an extensibility policy model for extensible operating systems, defines important basic policy concepts and suggests practical policy support for user-level policies. It also establishes a notion of a *kernel-oriented policy* to augment the University of Washington's security support for kernel extensions and implements it by extending domain and type enforcement to software segments. The third section discusses the SPIN security architecture developed at the University of Washington as well as architectural enhancements introduced under the EXOS project. The fourth section of the report examines how the identified system-level policy based on domain and type enforcement can be correctly extended in a *conservative* fashion to support system, middleware, and application-level services that must extend the operational security policy to support their security objectives. The fifth section describes how project-developed prototype demonstration software works and can be used to illustrate the security properties of the security-enhanced SPIN system. The sixth section presents conclusions and lessons learned, possible technology transfer, and recommendations for future work. We end the paper with a brief summary.

1 Introduction

A recent thrust of DARPA-funded operating systems research has involved approaches to system extensibility through user-application-initiated dynamic kernel augmentation. Example systems include SPIN [3] and the Exokernel [6] architecture, although each has major conceptual differences in terms of the fundamental approach that was adopted to system extensibility and the research emphasis of the work. Research on dynamic kernel extensibility is largely motivated by a desire to significantly improve overall application performance by better satisfying application-specific computing and communication requirements. A secondary motivation is the desire to develop a more flexible platform for exploring new OS principles and innovative concepts in operating system services, including security.

1.1 Goals and Approach

The primary emphasis of this project involved developing a practical approach to extensible operating system security including establishing the rationale upon which the approach is based. It specifically focused on the SPIN extensible operating system. SPIN introduces a level of operating system extensibility not previously encountered in operating system development efforts. SPIN core functionality is characterized as little more than a mechanism for loading and executing extensions.

Our work addresses the security needs surrounding the loading of user-developed extensions directly into the kernel. It also addresses the problem of least privilege among service components of the SPIN system that extend the functionality of the base operating system. Our approach integrates traditional security requirements with SPIN operating system development objectives, specifically high performance and fine-grained extensibility.

1.2 Security in Extensible Systems

Before discussing the security requirements raised by SPIN's pervasive system extensibility, we must clarify what is different between more traditional operating systems and extensible operating systems. As discussed in [15], most OSs are extensible to one degree or another, primarily to support functional enhancement. They allow for static or even dynamic linking of specific types of system components such as protocol stacks or device drivers to pre-defined interfaces. However, in typical operating systems this extensibility tends be along well-defined boundaries and restricted to specific areas of functionality. While developer support for such kernel extensibility has tended to increase in more recently deployed systems to augment dynamic configurability, it is still primarily constrained to drivers and protocol stacks.

An *extensible operating system*, as we employ the term, allows much more extensive alteration of system functionality than generally supported. It is fundamentally different from the traditional model in scale and approach. Extensibility extends to traditionally core OS services and allows the run-time loading and execution of operator- or application-supplied extensions to the kernel that extend or replace basic kernel functionality, including fundamental access control mechanisms such as virtual memory support. Moreover, extensibility may be supported at as fine a granularity as the replacement or customization of an internal kernel procedure call via user-supplied code.

This project has focused on SPIN system extensibility. SPIN extensibility support offers two significant benefits:

- the potential for improved application performance through the loading of application modifications or extensions into the SPIN kernel, and

- finer-grained security mediation.

SPIN employs kernel extensibility as a mechanism for improving OS performance in support of application software. A significant detractor from the performance of traditional operating systems is the overhead involved with supporting hardware-enforced execution contexts and switching between them. However, SPIN depends primarily on language features rather than hardware mechanisms for its safety features. In SPIN, procedure calls replace hardware context switches, reducing the overhead of supporting separate contexts, thereby contributing to improved system performance.

SPIN extensibility mechanisms also allow fine-grained security support. Software mechanisms, as employed in SPIN, can more easily provide protection at a finer level of granularity than hardware typically can. For example, hardware protection may define access enforcement at page boundaries, when what is required is access enforcement on non-contiguous memory regions. The latter can be provided by SPIN.

However, in spite of the flexibility and granularity of software-based protection mechanisms, they raise concerns among security professionals, who have been relying on hardware protection mechanisms for three decades. A convincing argument is required that the granularity, performance, and utility of software protection mechanisms outweigh the benefits of simpler, better-understood, and more widely accepted hardware protection mechanisms.

1.3 The Trusted Computing Base in Extensible Operating Systems

An issue that must be resolved with systems such as SPIN is that dynamically extensible kernels constitute a significant deviation from the traditional interpretation of operating system security principles and established high-security architectural concepts. Common security practice for high-security systems includes the operating system kernel in the definition of the trusted computing base, isolating it from user-level code through virtual address space management and hardware-enforced separation. Evaluating the kernel as a static code element has been one of the major elements of work in establishing an assurance level for a system.

The SPIN system, however, replaces the concept of a relatively static kernel with that of a dynamic kernel specialized at run-time to match system and user requirements. It supports dynamic, run-time alteration or augmentation of kernel functionality through untrusted as well as trusted extensions to a minimal "core" kernel code base that does not include much of the functionality we have come to expect in the traditional kernel code base. It introduces the problem of code that lies outside a trust boundary but inside the kernel. This raises the issue of how one defines and preserves an encompassing and non-bypassable security policy for the system. Also at issue is what security policies can be enforced by and on such systems.

2 Security Policy in the SPIN Extensible Operating System

Highly extensible systems manifest classical security requirements as well as introduce new ones. With respect to classical security requirements, user-level security needs still exist and must be addressed. Access control between user-level subjects and objects must be supported by and captured within a policy model that reflects the security objectives of the system.

However, extensible systems also introduce a new wrinkle on security policy support. Extensions loaded into the kernel may retain their association with user identity and user-level security attributes, such as security labels or type assignments. At the same time they become part of the OS kernel code footprint. As they are moved into the kernel's address space they acquire privileges associated with system-level functionality, despite the fact that they are of user-space origin and may be untrustworthy. This requires additional protection strategies to ensure that they do not violate both the explicit and implicit policies that apply to the user security context from which the code was introduced.

Extensible operating systems require that we consider the ramifications of the propagation of user-level security requirements across the user/kernel boundary. User-supplied extensions, running in kernel space, must be prevented from bypassing or otherwise subverting core kernel mechanisms that support policy. This motivates the development and inclusion of functionality that supports what we refer to as kernel-level security and requires that we establish and clarify the scope of kernel-level and user-level security and any overlap that exists.

The EXOS project investigated the means by which the analogue of user virtual address space protection might be efficiently applied to the SPIN kernel. There are two aspects to this kernel-level support, *confinement* and *least privilege*. Confinement separates application/system components into protected compartments not based on user identity. The purpose of confinement is to separate functionality into independent system components so that failures in one component are confined to a specific region of code and system functionality. It allows the developers to ensure that the operation of critical code segments is not subject to inadvertent or deliberate tampering by misbehaving code segments. In the case of extensible operating systems, we believe that it is especially important to implement some sort of extension confinement mechanism to ensure the integrity of system components.

The ease of extending the SPIN kernel underscores the need to restrict the privileges of extensions to only those necessary to support their intended functionality. System structuring to support *least privilege* was, therefore, a significant security objective for our extensible system security architecture. The objective was to provide a partitioning on the privileges available to

software components so that each system component operates with only the privileges required to support the intended service.

A fundamental concept explored in the EXOS project involved the use of security domains to provide a supplemental access control mechanism that is orthogonal to the user or authority that spawns a task. In traditional systems, a common model is that of a thread executing on behalf of a user request. Access control decisions are made based on the user's identity. However, in the SPIN extensible OS security architecture, there are the additional security objectives of least privilege and confinement. As the thread executes, it may progress through several different extensions, each of which has different security requirements. For example, a thread may execute device driver code that has special privilege and needs to be protected from tampering, or it may execute server code that has no special privilege but still needs to be protected from outside tampering. Hence, there is a requirement to provide security separation between these extensions in addition to the core system functionality itself. Security domains can be utilized to constrain the allowed and denied access modes associated with the thread along its locus of execution.

2.1 Policy Models

While there has recently been increasing interest in non-specific security policy expression languages such as Adage [19], a basic premise of the EXOS project is that there is significant value in the use of policy-specific models when it comes to stating the security policies that hold for a given system. The existence of a body of knowledge associated with specific policy models aids understanding of their implications in a given system and provides a context in which policy components can be considered and against which system design and implementation details can be reviewed. Furthermore, existing policy models reflect established application protection needs and provide useful abstractions for defining and understanding system security requirements. These policy models are widely recognized as addressing important government as well as commercial security requirements.

Part of the EXOS project work has involved supporting specific well-known user policy models to establish how important security policies can be supported by the kernel through SPIN policy specification and enforcement mechanisms. Several classical user-level security policy models including MLS, RBAC, Chinese Wall, DTE, and ACL based models were explicitly supported by the EXOS project under the SPIN security architecture and a policy model framework developed that accommodates inclusion of additional models.

Basic Concepts and Primitives

Prior to discussing the specific policy models supported, it is useful to define some basic security policy primitives included in the models: security objects, security subjects, and security domains, and how they relate to one another.

Security Objects

Security objects are identifiable, distinguished entities, including code segments and data containers, to which an active security policy applies and that can be recorded with a policy manager. They must have an associated set of security attributes by which enforcement

mechanisms mediate access by security subjects. Thus, not all system objects are necessarily security objects. However, if relevant policy-enforcement-critical objects are excluded, the security architecture of the system may be ineffective, providing a means to bypass a desired policy.

A security type is assigned to every security object. The security type of an object represents certain security attributes that are associated with an object. This type assignment groups objects that will be treated identically for access control purposes.

Security Subjects

A security subject is a security policy entity for which access to security objects is mediated by active security policy elements. The traditional definition of a security subject as defined in the "TCSEC Orange Book" is that a subject is a process-domain pair [5]. The SPIN system equivalent of a process is a SPIN thread, so we define subjects to be thread-domain pairs. As in the traditional case, a thread acts on behalf of some user. Threads move between security domains, the access to which may be constrained by the active security policy. In the SPIN security architecture, a thread initiates domain transitions by either executing a procedure call in kernel space or by executing or returning from a system trap to move from user space to kernel space.

Security Domains

A security domain may be both a constituent part of the security subject, as discussed above, and a security mediation target, depending on its use during any given mediation event. In the former case, it is characterized as a set of access permissions to objects. In the latter case, it is the mediation target of an access check event involving a subject's right to extend the domain or transfer control into that domain. In the EXOS project, we employ security domains to enforce the isolation of system-level components from one another, confining the effects of potentially misbehaving or subverted system components. Security domains are used to help preserve the integrity of the operating system by structuring the OS so as to minimize privilege and isolate components from one another.

Policy Model Support

For SPIN system security we were interested in two types of policy models--those that are based on the identity of the user associated with a thread of execution and those that are based on the execution locus of the thread. In the SPIN security implementation, we have associated the first of these, proxy-based models, with our support of user-level security requirements and the second, domain-based models, with our support of kernel-level security requirements. One area of investigation under this project was the degree to which these models, when simultaneously active in a running system, might potentially interfere with one another.

In this section, we provide an overview of specific security policy models for which we defined supporting policy language elements. These policy models encompass:

- Multi-Level Security Model (MLS)

- Role-Based Access Control Model (RBAC)
- Chinese Wall Security Model
- Access Control Lists (ACLs)
- Domain and Type Enforcement (DTE).

We have chosen these security models based on their apparent importance to the security community. We consider the first four under a discussion of what we refer to as *proxy-based policy models* and distinguish them from the DTE policy model. While it has been characterized as well as utilized as user-level security support, we feel that the DTE policy model is more applicable to system security objectives such as confinement and least privilege. We discuss the DTE policy model in the context of *domain-based security policy models*.

Proxy-based Policy Models

Proxy-based security policies encompass those security policies associated with the security identity under which a user-associated proxy, i.e., a process or thread, executes. The security attributes of the proxy are relatively static in nature, having been acquired when the proxy was created. They are intended principally to control the user's access to security-sensitive containers of information. Proxy-based security is ultimately based on some aspect of user identity or user privilege delegated to create proxies that reflect the user's access rights in security mediation decisions. These might include the groups to which the user belongs, the clearance level at which he is operating, or the role that he has assumed.

A user's access privileges are often stated in terms of an allowed range of access, rather than a unique access level. Consequently, the delegation of access privilege to a user proxy, such as a thread may, for example, require fixing of the proxy's security label at one of several allowed labels contained within the full label range associated with the user's identity. For example, a subject created on behalf of a user will be only be given an MLS label allowed by his MLS maximum-read-level and minimum-write-level. Similarly, a subject created on behalf of a user will only be given a role that is permitted to the user. It is the specific label or roles attached to thread or process proxies rather than those potentially allowed by user identity, which we are dealing with in proxy-based policy support.

Multi-level Security Model

The multi-level security (MLS) policy model defines lattice-based information flow control based on the sensitivity of information. The goal of an MLS policy is to prevent information from higher sensitivity containers from flowing into less sensitive containers. MLS policy support requires the identification of security subjects, associated with users and their security clearance, and security objects, associated with containers of information. A (clearance, category-set) label is attached to each subject consistent with the read-max and write-min constraints of the initiating user and a (level, category-set) label to each object. The classification levels establish an ordering that represents the relative sensitivity of information identified with that level and the policy defines constraints on the flow of information between such levels.

The sets of categories that are assigned to objects are also used to order objects by their sensitivity. An object or data container labeled with a set of categories and a certain classification

level is considered more sensitive than an object or container labeled with a *subset* of those categories and the same classification level. The former object's label is said to dominate the latter's. Therefore, the MLS policy model requires the assignment of sensitivity labels to subjects that represent the maximum sensitivity of information that they are allowed to read. The MLS policy states that a subject is only allowed to read information labeled with a sensitivity label that is dominated by the label of the subject (simple security property) and write information labeled with a sensitivity label that dominates the label of the subject (*-property).

Role-based Access Control Model

RBAC establishes work roles in which the user may operate and controls access to information based upon those work roles. It is the role that the user has assumed for an interval of execution that determines what security objects he can access. Initially, RBAC policy models simply identified the collection of supported roles and the user's ability to assume a particular role or set of roles during a computational session and bound object access rights to these roles. More recent work, the RBAC96 model [13], goes beyond simple role identification in an attempt to support hierarchical role relationships as well as a family of RBAC models. Sandhu [14] identifies a three-tier architecture comprised of four separate but related models for specification and enforcement of role-based access control. $RBAC_0$ is comprised of users, roles, permissions, and sessions. $RBAC_1$ introduces a concept of role hierarchies to the $RBAC_0$ model analogous to object hierarchies in an object-oriented model. $RBAC_2$ is incomparable to $RBAC_1$ and adds a concept of constraints on roles to the $RBAC_0$ model. For example, constraints may indicate that the same user cannot belong to two distinct roles; i.e., they are mutually disjoint roles. $RBAC_3$ represents a consolidated model that combines $RBAC_1$ and $RBAC_2$ to provide both role hierarchies and constraints.

RBAC96 makes a distinction between users and sessions. Roles active in a session can be changed at the user's discretion. However, RBAC96 views the changing of roles as a security-sensitive act that constitutes a relatively heavyweight event that directly involves the user (although the model does acknowledge the concept of one session creating another session while not speaking further to that issue). We, too, view the transition between roles or activation of roles as a heavyweight event minimally associated with the creation of a new task, and not a control flow transition from one domain to another. In other words, we, at a minimum, see a new thread or process being created to represent the new role association, if not an even more substantial event such as a new logon action.

In the SPIN security architecture we restrict our policy language and security implementation to Sandhu's $RBAC_0$ model. Support for Sandhu's $RBAC_1$, $RBAC_2$, and $RBAC_3$ models is covered by role-specification and assignment mechanisms, of more of an administrative character, unrelated to the underlying mediation and enforcement mechanisms we are investigating for the SPIN system. They do not require any special run-time support beyond the role labeling established for $RBAC_0$ support. We considered them outside of the scope of the EXOS project.

Chinese Wall Model

The Chinese Wall security model addresses the problem of upholding the confidentiality of information to which an analyst or consultant might be granted access in the course of working

for his clients. It is characterized in the literature as the *code of practice* that must be followed by a consulting company. For example, a consulting company may have contracts with two companies that are in direct competition with one another. If no special policy is in place to handle insider knowledge, the two companies may both fear that the consulting company is a conduit of proprietary information.

The Chinese Wall model represents the application of this type of policy to computation. In this policy model, information is grouped in three different ways. At the lowest level, the individual containers of information are called objects. At the next level, objects are labeled with the company dataset to which they belong. These company datasets represent the information that is associated with a particular company. At the top level, company datasets are grouped together into conflict of interest classes (COIs). If two different company datasets are in the same conflict of interest class, then it is viewed as a conflict of interest for a user to access objects from both datasets. The statement of the Chinese Wall policy is that a user can obtain access to objects in only one company dataset for each conflict of interest class.

We have selected the model described in [12] as a basis for our policy language primitives for Chinese Wall. This model assigns a label to each user, his run-time-instantiated subjects, and all COI security objects in the system. The label is defined as a vector of identifiers associated with company datasets. Each position in the label vector corresponds to a particular conflict of interest class. In the case of the user label, a given position in the label vector contains the identifier for the company dataset accessed under that conflict of interest class, or NULL if no dataset has yet been accessed.

The subject label indicates the access rights allowed during a given session, and is constrained by the access history recorded for the user label when the subject's label is created. The object label indicates the COI classes and associated datasets to which the object belongs (i.e., the object contains data from the indicated datasets associated with corresponding COI classes). Since the Chinese Wall policy model allows access to only one member of a COI class for a given user or his surrogates, the user label vector represents the history of user access to COI class datasets while the subject label constrains the run-time access rights to company datasets.

The intent is that a user be able to access the data associated with new companies within the constraints of other access controls provided that the user does not obtain access to two companies that have a conflict of interest. Thus a new user will be labeled as having never accessed any of the company datasets within the specified COI classes. A user's label floats upwards in the lattice with respect to dominance of subject/object labels as he is allowed access to datasets within COI classes.

A user interacts with the system by creating a subject to act on his behalf. When the user creates a subject, the user specifies a label for that subject that indicates what company datasets the subject is allowed to access, constrained by the user's access history and any discretionary security policy controls that apply to the user. As a subject attempts to access a dataset within a specific conflict of interest class, a check is made to see if and how a dominance relationship (defined on the subject's and object's label vectors) is satisfied with respect to the granting of specific combinations of read, write, and append access from the subject to the object.

ACL-based Model

An ACL-based policy provides controls over how a subject can access a named object; control is at the discretion of users. An access control list is simply a list of principals that are authorized to have access to some object along with the allowed access rights. As stated in the Orange Book: *These access control lists shall be capable of including or excluding access to the granularity of a single user* [5]. Access control lists may specify *allowed* access, *denied* access, or a combination thereof. An implicit ordering exists as to the precedence of identity type entries. An implicit rights composition rule also exists as to how expressed *allowed* and *denied* access permissions are combined from multiple applicable ACL entries to determine the actual access permissions associated with a target object. Conflicts between allowed and denied access permissions require an ACL search rule or some other criteria that establishes how such conflicts are resolved. Actual interpretation of access rights depends on the kind of object with which the ACL is associated.

Domain-based Policy Models

Domain-based security policy support introduces an additional security constraint on the security subject beyond those we characterize as proxy-based constraints. Domain-based policies apply a dynamically variable security attribute in mediation decisions representative of the *domain of execution* in which a thread is executing. This domain-based security constraint is actually utilized in three different types of security decisions: the objects that a subject can access from a given domain (similar to traditional proxy-based security support except that it may involve access control to both system-level as well as user-level objects), the domains to which the subject can transition (i.e., other access rights it can acquire), and actions it can invoke against other domains.

Past as well as recent secure operating systems work [1,7,16,17] has utilized the domain concept as a mechanism for isolating system components from one another. The EXOS project adopted the concept of security domains as being particularly well-suited to extensible operating system security to insure that misbehaving extensions cannot compromise other kernel or application functionality.

DTE Policy Model

The DTE policy model is based upon the Type Enforcement model first proposed by Boebert and Kain [4] for the Secure Ada Target (SAT) and subsequently implemented in the LOCK program [16]. Under Type Enforcement, each security object is assigned a *type*. Domains of privilege are established in which programs execute. Access to specified types is restricted to specified domains. DTE also supports the notion of control transfer between domains, defined at the object-module level, through specifically identified domain transition points associated with individual domains. These domain crossing points are identified statically in the DTEL language policy specification.

The TIS/NAI Labs DTE work [1] remedies a particular difficulty associated with Boebert and Kain Type Enforcement, while supporting the underlying policy model. The tables used under Type Enforcement to express domain-to-type and domain-to-domain access rights can grow quite large and cumbersome to represent and process on each access-mediation event.

Recognizing that named, persistent objects, e.g., files and directories, make up the bulk of the security objects in traditional operating systems such as UNIX, DTE refines the techniques of Boebert and Kain by defining a sparse, shadow directory tree with parent-node-based default type inheritance. Explicit type inheritance can be used to override the default, reducing the storage and processing requirements associated with the technique. DTE also specifies a relatively simple language for specifying type-enforcement-based security policies.

Domain/Type Enforcement and SPIN

While DTE has been utilized to explicitly support proxy-based access control policies such as simple role-based access control, we found it most useful in the expression and support of confinement and least privilege requirements attached to extensions. Past experience with other secure operating systems work suggests that user extensions to the kernel should not all live and execute in the same protection domain even with software guarantees such as the strong data typing supported in the Modula-3 language used to implement SPIN. The domain concept is especially applicable to extensibility-oriented policy support and corresponds well to both the confinement of untrusted physical code segments, such as user-loaded extensions and the minimization of privilege required by specific functionality. If we associate kernel extensions (and possibly static kernel functionality) with particular domains utilizing type assignment to reflect domain-based access control requirements, DTE allows the imposition of a level of structuring on the core OS that isolates the effects of aberrant system behavior.

We implement security domains through the identification of gates that effect a transition from one security domain to another. This functionality was already present in the University of Washington's security enhancements to the base SPIN system functionality. However, we went further by associating a collection of *typed* kernel entities with security domains, specifically, software segments to which individual modules are assigned so as to establish desired system structuring and confinement properties. Software segments allow the compartmentalization of kernel code into separate sections with common functional and protection attributes. They allow the system to determine the domain in which a thread is executing before any security-relevant procedure call. They permit mediation and enforcement of access control decisions when a security-relevant procedure call is made.

3 EXOS Architectural Enhancements

This section details the enhancements made to the original SPIN security architecture in order to support some specific user-based and system-based security policies. Elements of the full SPIN system architecture can be found in [15]. This discussion begins with a description of the original SPIN architecture, with an emphasis on the original security mechanisms, and continues with descriptions of the enhancements made to the SPIN policy manager under the EXOS project to support the enforcement of specific user-based policies. The section concludes by detailing enhancements made to support the enforcement of system-based policies through software segment access control, which is also mediated by the policy manager.

3.1 Original SPIN Architecture

The SPIN kernel is conceptually a single program written in the object-oriented programming language Modula-3, in addition to a stand-alone layer of code for low-level control

of the target hardware. Higher-level kernel components include Modula-3 runtime support (strong type support, low-level linking, and garbage collection support) and several core services. The core services include memory management, an event dispatcher, and a strand scheduler, in addition to those that will be emphasized in this section--the dynamic linker and the enforcement and policy managers.

SPIN Domains and Dynamic Linking

Because SPIN allows applications to reside in the same address space as the kernel, many mechanisms must be employed to protect kernel code and data from unwanted tampering and execution. At a low level, SPIN employs compiler restrictions on the extension code written in Modula-3 that preclude unsafe pointer operations such as pointer arithmetic and casting. This ensures that extension code cannot directly access data or code that has not been provided to the extension at link time.

The SPIN kernel also needs to control the access that extensions are given at link time. This is done through the concept of a SPIN domain [11]. The *SPIN domain* represents the unit of linkage in the SPIN system. It encapsulates a set of interfaces or a body of code that is to be linked against. The SPIN domain is represented as a Modula-3 object that cannot be forged by SPIN extensions because of the type-safety constraints. By treating a SPIN domain as a capability and controlling how SPIN extensions obtain access to SPIN domains, the SPIN kernel can restrict which extensions obtain access to privileged kernel calls.

It is important to observe that SPIN domains are different from security domains. SPIN domains are passed as capabilities. Once a SPIN domain is given to an extension the extension has full control over linking to the SPIN domain. Thus SPIN domains are not subject to any mandatory policy such as a domain/type policy.

Access Control

SPIN's runtime access control mechanisms are the major responsibilities of two entities: the enforcement manager and the policy manager [9]. The role of the enforcement manager is to intercept every security-relevant access attempt and to request a mediation decision from the policy manager when necessary. The enforcement manager is policy neutral--it does not interpret subject or object identifiers or access modes. It simply provides access enforcement as requested by the policy manager and other policy-aware SPIN components.

When a SPIN domain requires access control on an exported procedure, it notifies the enforcement manager of this fact, along with the access modes that are required for procedure execution, access to the procedure arguments, and the result of the procedure. In addition, the SPIN domain may specify whether execution of the procedure signifies a SPIN domain transition for the subject, and, if so, the new SPIN domain.

When a procedure execution is attempted for the first time, the enforcement manager notifies the policy manager with the security identifiers (SIDs) of the subject (thread/domain pair) and the procedure to which access is to be mediated. The policy manager then returns the appropriate access modes for the subject/procedure pair to the enforcement manager. The subject/procedure call pair, along with the access modes, may be cached if requested by the policy manager. If the returned access modes include the mode required for executing the

procedure, each of the procedure arguments is tested in a similar manner. If those tests pass, and a SPIN domain transition is specified for the procedure, the SPIN domain is changed and the procedure executes. On return from the procedure, the SPIN domain is possibly changed, then the return value tested.

3.2 Implementing a Policy Manager

The goal of the University of Washington SPIN security work was to produce a policy-neutral enforcement manager. Therefore, the policy manager provided by the University of Washington was intended primarily for testing purposes. Part of the goal in our work was to build a prototype policy manager to mediate several real-world policy models. Our policy manager utilizes a policy interpreter, which reads in the policy specification file and initializes the policy manager, and policy representations, which effect mediation of the policy. In addition to the description of the policy manager, the algorithm for mediating policy is described below.

Policy Interpreter

When the SPIN system boots, the policy interpreter reads in the policy specification file. In the present implementation of SPIN, this file is obtained from the SPIN build machine. The policy specification file is a plain text file, and is transmitted to the SPIN crash machine via the trivial file transfer protocol (tftp).

The policy interpreter first checks the syntax of the policy file. If an error is found in the syntax, the policy interpreter outputs a descriptive error message to the SPIN console and terminates. If the syntax is found to be correct, the policy interpreter populates the data elements of the policy manager that represent the security policy itself.

The policy interpreter is loaded at SPIN boot time as a trusted extension to the SPIN kernel. Parser and scanner functionality are written using the popular compiler/interpreter tools yacc and flex, and the actions that populate the policy manager data structures are written in C. There is a separate facility that allows checking the policy specification file for syntax errors before actually booting SPIN. The policy specification file is written in a language with elements that closely correspond to the descriptions of the individual policy representations described below.

Policy Manager Internals

Before discussing the implementation of the policy manager, it is instructive to describe the overall design of the elements interior to it, including the SID/attribute table, a mapping between the security identifiers, SIDs, of security-relevant entities, and the attributes associated with those entities. Also detailed in this section are the data structures that the policy manager uses to represent the policies to be mediated. In addition, we detail the attributes that are associated with objects' and subjects' SIDs in the SID/attribute table, and that are used in the policy representations to make policy decisions.

The design of the policy manager is shown in Figure 1. The data structures interior to the policy manager include a mapping between SIDs and attributes. This table is designed as a hash

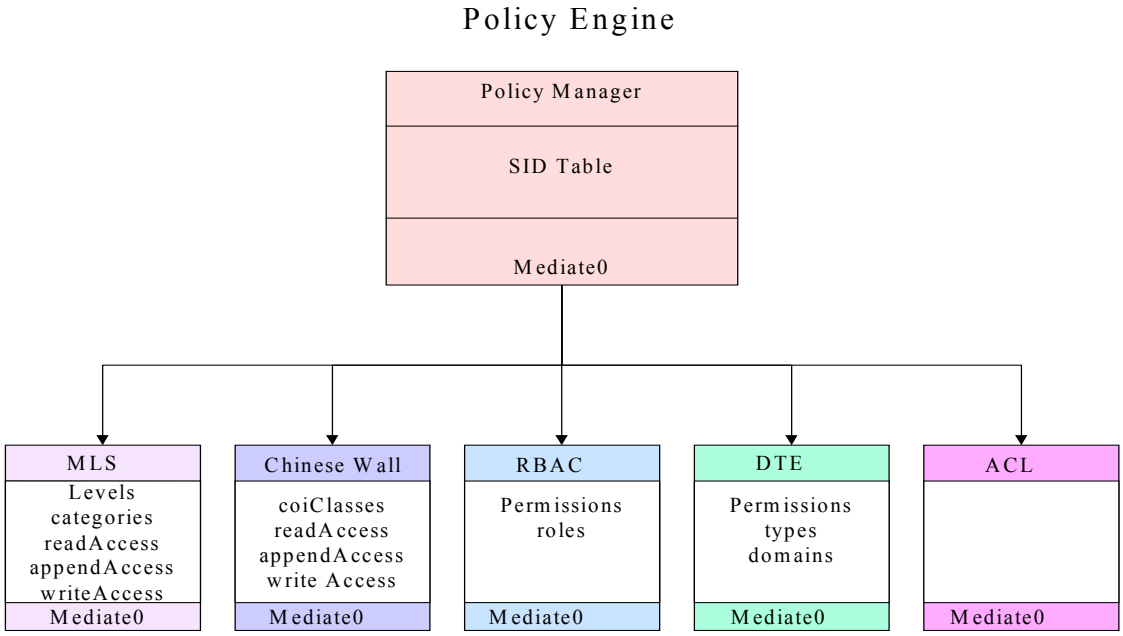
table with the SIDs as keys for quick lookups when the policy manager is queried with the SIDs of a subject and object.

The policy manager includes representations of five different policy models: MLS, Chinese Wall, RBAC, DTE, and ACL. These representations contain policy-specific information that is independent of the system security entities. The MLS representation contains a list corresponding to the security levels present in the system along with a list of categories to which datasets may belong. The Chinese Wall representation contains the datasets belonging to the conflict of interest classes. In addition, both the MLS and Chinese Wall policy representations contain lists of access modes that correspond to read permission, write permission, and append permission. In the MLS and Chinese Wall policies, these are the only access modes allowed. The RBAC representation contains a table of roles. Each role associates the name of the role with a list of permissions, and each permission is a <object name, access mode list> pair. In addition, the RBAC representation keeps a list of permissions so that they may be accessed quickly. The DTE representation also contains a list of permission objects for quick lookup, as well as a list of domains. Each domain consists of a name, a list of permissions corresponding to the name, a list of privileges, and a list of components. Each permission is a <type, access mode list> pair. The ACL representation has no data structure associated with it because it needs none to represent its policy model. All policy-relevant information for the ACL model is contained in the ACL attributes carried along with the associated object representation.

Each of the policy representations makes mediation decisions based on the subject and object attributes that it is passed by the policy manager. The structures of these attributes depend on the policy representation in question, and on whether the attribute describes the subject or object involved in the mediation. Before describing mediation for each of the policy representations, then, it is necessary to describe in detail the structure of the attributes involved.

The MLS subject and object attributes keep the level and a list of categories to which the subject has access or to which the object belongs. The Chinese Wall subject and object attributes keep a vector of the datasets in the conflict of interest classes that the user has visited, or to which the object belongs. The first element of the vector corresponds to the first conflict of interest class, the second element to the second COI class, etc. An element of the subject attribute vector may contain a single dataset label if the user has visited the dataset, or it may contain a NULL if the user has not visited any dataset in the corresponding conflict of interest class. Similarly, an element of the object vector may contain either a single dataset or NULL.

The RBAC subject attribute contains the roles to which the subject belongs, and the RBAC object attribute contains a label corresponding to the name of the object (see the above discussion of the RBAC representation). The DTE subject attribute contains the domain in which the subject exists, and the DTE object attribute contains a label corresponding to the type of the object. Finally, the ACL subject attribute contains a label corresponding to the name of the subject, while the ACL object attribute contains the access-control list, a list of subject names and corresponding access modes.



Policy Manager Mediation

As detailed in the discussion of the policy/enforcement manager protocol, the main responsibility of the policy manager is to return a set of access modes based on the SIDs of the subject and object in question. In order accomplish this, the policy manager, when it receives a mediation request containing the subject and object SIDs, first looks up the attributes corresponding to the SIDs in an internal table. After obtaining the attributes, the policy manager then calls each active policy representation (one or more of the MLS, Chinese Wall, RBAC, DTE, and ACL representations) with the parts of the attributes relevant to them; i.e., it calls the MLS representation with the MLS subject and object attributes, the Chinese Wall representation with the Chinese Wall subject and object attributes, and so on. Each representation takes the subject and object attributes and returns a set of access modes reflecting the rights that the subject has to the object. The policy manager then collects the results and returns the intersection of the access modes. With this procedure in mind, then, it is instructive to describe the mediation of the individual policy representations.

The MLS and Chinese Wall policy representations have very similar implementations. They both utilize a dominance relation between subject and object attributes. The details of these dominance relations are different for the two policies and are described below. However the manner in which the dominance relation is used for mediation is the same for the two policies. If the subject attribute dominates the object attribute, then the Read access mode is returned. If the object dominates the subject, the Append access mode is returned. If they are found to be of equal dominance, then the Read, Write, and Append access modes are returned. Note that these composite access modes are mapped to atomic access modes in the policy specification, so that a set of atomic access modes is returned in each case. It is possible that all of these tests fail. In that case, the attributes are incomparable, and an empty set of access modes is returned.

For the MLS policy representation, the subject dominates the object if the level of the subject is greater than or equal to the level of the object and the category set of the subject is an improper superset of that of the object. The object dominates the subject if, in the previous condition, the subject and object MLS attributes are reversed. If the subject and object have equal levels and categories then the subject and the object have equal dominance. If all three of these tests fail, the MLS attributes are incomparable.

For the Chinese Wall representation, the subject and object vector elements are tested one by one. The subject dominates the object if, for each vector element, the subject's element is equal to the object's element, or if the object's element is NULL. The object dominates the subject if, in the previous sentence, the roles of the object and subject are reversed. If all of the subject and object vector elements are equal, the subject and object have equal dominance. Again, if all of these tests fail, the attributes are incomparable.

The RBAC representation makes mediation decisions by taking the object's RBAC attribute, and searching through the subject's roles collecting the access modes corresponding to the object's label. After collecting the access modes, redundant modes are eliminated, and the resulting mode set is returned. The DTE representation works in similar fashion. Given the object's type label, a search through the subject's domain determines the access modes corresponding to the type. This set is returned. The ACL representation takes the subject's attribute and searches for the label corresponding to the subject in the object's ACL. It returns the access modes corresponding to the subject's label.

3.3 Segmentation Enhancements

As a complement to the policy manager enhancements described above, segments and associated access controls were added to SPIN in order to support least privilege and confinement. We view a segment as a SPIN security object, and a thread executing in a particular domain as a security subject. By enforcing a security policy on the subject/object combination, segments can be protected against illegal or unwanted reads and writes. In addition, threads attempting to execute code in a segment can be prevented from doing so.

Segments

A *segment*, as we define the concept, is an association of related Modula-3 modules with common privilege and confinement mechanisms that are treated as a single entity for protection purposes. In implementing segments, we borrowed somewhat from the similar Multics [10] concept; however, we employ a segment as a logical subdivision of the kernel's address space based upon Modula-3 module lines. We do not require hardware support for segments, although we do exploit hardware features (described below) in order to facilitate the implementation of segments and associated access control, and in order to improve performance. With the proper hardware support (such as exists on the Intel x86 platform), segments could correspond directly to more traditional hardware segments. We chose to implement a more flexible solution while retaining some performance advantages of using hardware segments.

Segment Access Caches

Segment access caches keep track of access decisions that have been made by the segment manager, described below. There is one segment access cache per subject in the system. At any given time, the current access cache is the segment access cache for the currently executing subject. The intention is that the segment access cache can be quickly accessed when an access operation is attempted. When an access operation is first attempted, the cache entry will be marked invalid and the operation will fault. As a result of the fault a mediation procedure is called. If the access is granted then the cache entry is filled in with the appropriate access modes and the operation is allowed to continue. If the access is denied then an access fault signal is sent to the calling thread. When a context switch occurs as a result of a gate crossing or if the scheduler runs a new thread, the current segment access cache is changed to be the segment access cache for the new subject. The use of the segment access cache is very similar to the use of the translation lookaside buffer in hardware. In fact, on certain hardware platforms it may be possible to implement the segment access cache as the hardware translation lookaside buffer.

In our extension of the SPIN system, each segment access cache is composed of an array of four-bit words. The elements of the array correspond to the segments existing in the system. For each segment, the bits in each array element are one or zero according to whether read, write, or execute access was previously granted to the subject/object pair. In addition, one bit in each element is zero or one according to whether the subject/object pair has been mediated. For fast convenient access, the segment access cache is held in a hardware segment.

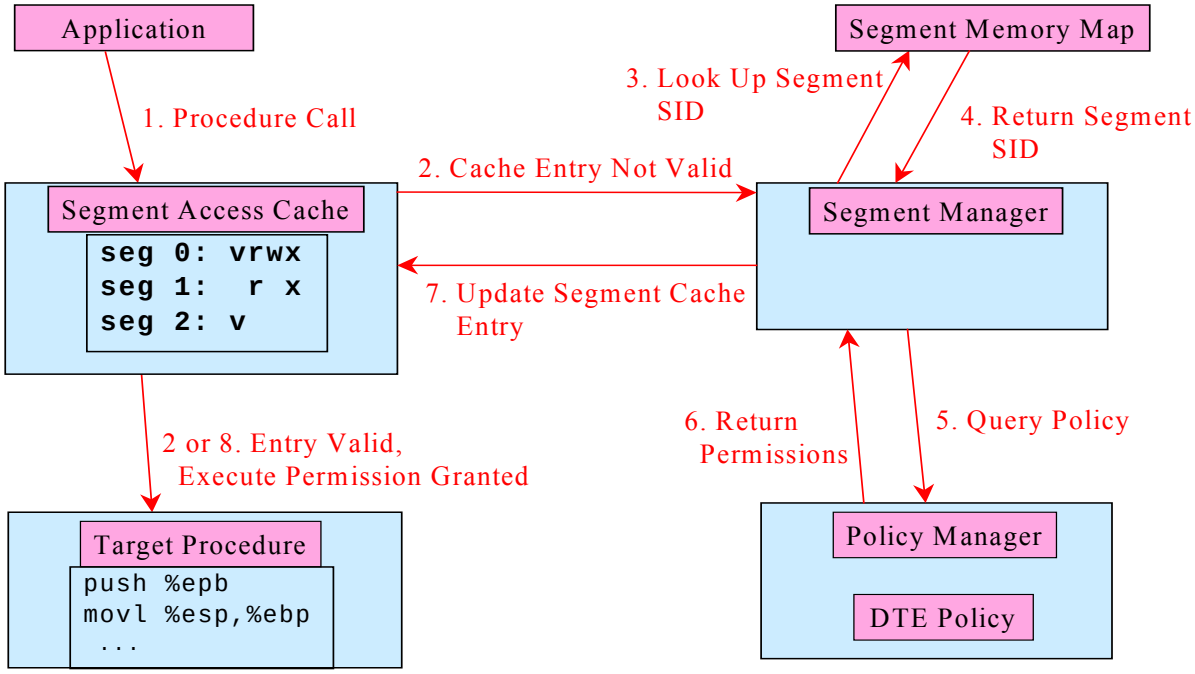
Segment Access Intercept Mechanism

The segment access intercept mechanism exists to ensure that a subject can read or write data, or execute code in a segment if it is permitted by policy. When a subject attempts to read or write data, or execute code, the segment access intercept mechanism extracts the appropriate segment access cache entry from the hardware segment register, and, if the valid bit is not set, a segment access fault occurs. If the entry is valid and the appropriate read, write, or execute permission is not given, the attempt fails and an exception is thrown. The segment access intercept code is inserted by the SPIN C/C++ compiler, which also serves as a back end for the SPIN Modula-3 compiler. Initially, when code is compiled, it is not known what segment the code is attempting to access. Only at link time will this be known; thus the linker patches the compiled code with the appropriate segment number, which is the offset into the hardware segment (the segment access cache entry).

Segment Manager

The segment manager has two responsibilities: to hold a mapping from address ranges to segments, and to handle segment access faults. The linker, when it encounters code that is attempting an access, looks up the relevant segment, and patches the code with that segment. In order to look up the segment, the linker queries the segment manager with the address of the code or data in question, and the segment manager responds with the segment in which the code or data exists. The other responsibility of the segment manager is to mediate segment access when a segment access fault is generated. To do this, the segment manager queries the policy manager with the domain/segment pair and updates the bits in the segment access cache entry, along with the valid bit. The access is then retried.

Segmentation Architecture



Domain/Segment Access Control Mechanism

The figure reflects the process that occurs when an application attempts a procedure call, but the process is similar for data read or write attempts. When the application attempts to call a procedure (1), the segment access intercept mechanism extracts the appropriate segment access cache entry from the hardware segment register. This register has been set to the current subject's segment access cache. The individual entry (for the procedure's segment number) is obtained efficiently via an offset into the hardware segment. If the valid and execute bits are set (indicating that execute permission has been granted previously for this subject/object combination), the call proceeds as usual (2a). If the valid bit is set and execute permission is not granted, an exception is thrown. If the valid bit is not set, the segment manager is consulted (2b). The segment manager first looks up the segment SID in the segment memory map (3,4), then queries the policy manager with the current subject and object SIDs (5). The policy manager calculates the permissions allowed the subject/object pair using the DTE policy, and returns the set of permissions to the segment manager (6). Using this set of permissions (which is a subset of read, write, and execute), the segment manager updates the segment access cache entry with the granted permissions, which may include read and/or write permissions in addition to the permission of interest in this example, execute permission (7). The call is then retried, and, in a manner similar to (2a), if execute permission is granted, the call proceeds (8). If execute permission is not granted, an exception is thrown. Note that, on all future calls or data reads and writes with this subject/object combination, the segment access cache entry is valid, and the call or access attempt is very efficient.

4 Policy Extension

In this section we explain the need for a mechanism that allows programs to extend the domain/type policy dynamically and describe how such a mechanism can be implemented. In addition, for illustration, we present two examples of the extension mechanism.

4.1 Approach

The domain/type policy has been proven very successful in confining a predefined set of programs to privileged but limited domains to enforce a least privilege requirement. For example, the DTE project [1] at TIS/NAI Labs has demonstrated how to confine programs that require root privilege so that they do not have all of the permissions associated with root privilege. For extensible operating systems, it is necessary to extend this capability to include policy constraints and protections for user programs whose needs were not foreseen at system build time. In order to satisfy the needs of such programs, we need a policy extension mechanism. In order for this extension mechanism to be useable by both untrusted applications and middleware, the extension mechanism must not allow the creation of new policies that are inconsistent with the intent of the old policy.

The major requirement of our approach is that the subject requesting the policy extension need not be trusted. Thus an extension is allowed only if the system can guarantee that the extension does not result in breaking guarantees that have been made to existing programs. For example, if a trusted program relies on the domain/type policy to prevent any other subjects from accessing a file, then this guarantee must still remain in place after any extension to the policy. We call such an extension *conservative*.

The technique that we use to meet this requirement is to require that both the original and the new policy constraints hold after a policy extension. When the policy is extended, every domain (type) in the new policy maps to a domain (resp., type) in the old policy. In addition, every event allowed by the new policy will map to some event that is allowed by the original policy.

4.2 Extending the Domain/Type Policy

We now define how a domain/type policy can be extended in a conservative manner. We start with a notation for representing domains and types that leads naturally to a method of introducing new domains and types. We then describe the three types of extensions that can be made to a domain/type policy and define the mapping of the new policy to the old policy. Finally, we describe how this technology can be applied.

Representing Domains and Types

The domains and types that are defined in the policy language are all atomic domains and types. The *atomic domains and types* are exactly those domains and types that are available when the system is first started, before any policy extensions have taken place. *Composite domains and types* are those domains and types that are built out of existing domains and types

when the system policy is extended. In order to define a notation for the composite domains and types, we assume that '/' is a character that does not occur in the representation of any atomic domain or type. A composite domain or type is represented through a path notation

$d_1/\ldots/d_n$

or

$t_1/\ldots/t_n$.

Here d_1 and t_1 are atomic domains and types respectively. For each k>1,

$d_1/\ldots/d_k$

represents a new domain that has been introduced through a policy extension. This new domain is based on the domain

$d_1/\ldots/d_{k-1}$

that existed before the extension. Similarly,

$t_1/\ldots/t_k$

represents a new type that has been introduced through a policy extension. This new type is based on the type

$t_1/\ldots/t_{k-1}$

that existed before the extension.

This notation has the property of maintaining a history of how new domains and types are introduced as the system policy is extended. At any given time, the domains and types that are available are those that either existed when the system was started or that were introduced by some policy extension. Initially, the only domains and types that are available are the atomic domains and types. Thus the domain, $d_1/\ldots/d_n$, is the result of n-1 extensions. In the original policy, the source of this domain was the atomic domain, d_1. The first extension adds a new domain of the form d_1/d_2. These extensions continue until the domain $d_1/\ldots/d_n$ is created.

Now we are ready to discuss the ways in which the system policy can be extended. There are three kinds of policy extensions that can occur. We can create new domains, new types, or new access modes.

Adding New Domains

We now describe how a new domain is added to the policy. Such a policy extension is made by adding a new domain of the form

$d_1/\ldots/d_k$

to a system that already has a domain of the form

$d_1/\ldots/d_{k-1}$

We will also need a rule constraining how the access matrix will be modified. The rule that must hold is that the access modes that are allowed for the domain $d_1/\ldots/d_k$ must be a subset of the access modes that are allowed for the domain $d_1/\ldots/d_{k-1}$. Any change to the access matrix that respects this rule is allowed.

Another constraint on the extended policy involves the authorization checks that must occur when a subject is created in the new domain. These checks ensure that an untrusted program cannot simply load some code into a domain that confers privilege without some type of check. The rule that must be followed here is that the checks required for loading code into the domain $d_1/\ldots/d_k$ must be at least as stringent as the checks required for loading code into the domain $d_1/\ldots/d_{k-1}$.

To show that this policy extension is conservative, we must demonstrate a mapping of the new policy and all the allowed events into the existing policy and the events that were allowed there. The key point that makes the definition of this mapping possible is that the allowed access modes for the domain $d_1/\ldots/d_k$ are at least as restrictive as the allowed access modes for the domain $d_1/\ldots/d_{k-1}$. Thus the mapping that takes the domain $d_1/\ldots/d_k$ to the domain $d_1/\ldots/d_{k-1}$ will map the events in the extended policy to events in the existing policy.

New domains are added to the system in order to confine a program that would otherwise have more privilege than it needs or to allow a program to create a domain for itself that will allow it to avoid tampering by attackers.

Adding New Types

New types are introduced in much the same way that new domains are introduced. Such a policy extension is made by adding a new type of the form

$t_1/\ldots/t_k$

to a system that already has a type of the form

$t_1/\ldots/t_{k-1}$

As with the extension that adds a new domain, we must have a constraint on the access matrix. The constraint is that the allowed access modes that domains have to the new type must be a subset of the allowed access modes that domains have to the type on which it is built.

New types are added to a system in order to provide a confined program with objects that it can access without damaging critical files or to give a program that is creating a protected subsystem objects that it can use that cannot be tampered with.

To show that this policy extension is conservative, we must demonstrate a mapping of the new policy and all the allowed events into the existing policy and the events that were allowed

there. The mapping that we will use maps the type $t_1/\ldots/t_k$ to the type $t_1/\ldots/t_{k-1}$. The mapping is conservative because the access that subjects can have to a type $t_1/\ldots/t_k$ is a subset of the access that subjects can have to the type $t_1/\ldots/t_{k-1}$.

Adding New Access Modes

New access modes are introduced when a new kind of object is introduced. For example, suppose a new extension is introduced that implements a distributed file system. This extension introduces a new kind of object into the system. It may be that this extension introduces new access modes for the new kind of object representing different cache coherency policies for reading and writing. This new extension may have different versions of the read and write access modes representing the different cache coherency policies. The rule that this mapping has to follow is that the new domain/type access matrix must be the same as the old access matrix except that the new access modes may be added in arbitrarily throughout the matrix to regulate access to the new object kind.

To show that this policy extension is conservative, we must demonstrate a mapping of the new policy and all the allowed events into the existing policy and the events that were allowed there. For this policy extension, the new kind of object has no analogue in the existing policy. Access operations in the new policy merely map to inter-process communication in the existing policy. Therefore, accessing the new kind of object with the new access modes maps to a no-op in the existing policy. Thus, the requirement that the access matrices differ only in that the new access modes are added to the access matrix is sufficient for the policy to be conservative.

4.3 Examples

We now give two examples of policy extensions. The first example involves the problem of confining a potentially hostile application to a domain where it cannot damage user files and applications. In order to keep the example simple, we start with a trivial domain/type policy that can be represented by the following access matrix:

	Initial Type
Initial Domain	rwx

This policy states that all subjects will live in a domain that has read, write, and execute access to any subject. We now have the problem that a new subject is going to be introduced that is potentially hostile. We do not want to put this new subject in the ‘Initial Domain’ because it would be able to corrupt arbitrary files in that domain. We therefore create a new domain and type.

We first create the new type ‘Initial Type/Corruptible’.

	Initial Type	Initial Type/Corruptible
Initial Domain	rwx	rwx

All objects of the new type will be susceptible to attack from the hostile subject when it is created. Note that the each set of access modes to the new type for each domain is a subset of the set of access modes for each domain to an old type, in this case, the ‘Initial Type’.

Now we create the new domain in which the potentially hostile subject will live:

	Initial Type	Initial Type/Corruptible
Initial Domain	rwx	rwx
Initial Domain/Hostile	-	rwx

In this new policy, we have a domain ‘Initial Domain/Hostile’ where we can place the new subject. From this domain, the new subject will have no access to objects of type ‘Initial Type’. It will in contrast have full access to objects of type ‘Initial Type/Corruptible’ which it can share with subjects in the domain ‘Initial Domain’. The set of access modes for the new domain to each type is a subset of the access modes for an original domain, ‘Initial Domain’, to each type. Similar examples can be constructed to illustrate protection of middleware and least privilege.

As a slightly more complicated example, we consider a device driver that needs access to some, but not all, kernel resources. We start with a policy that models two levels of privilege:

	Protected	Unprotected
Trusted	rwx	rwx
Untrusted	-	rwx

We now want to introduce a device driver that needs some privilege but not all the privilege needed by trusted applications. After two extensions ('Protected' to 'Protected/Device Resource' and 'Trusted' to 'Trusted/Device Driver') we arrive at the following policy:

	Protected	Unprotected	Protected/Device Resource
Trusted	rwx	rwx	rwx
Untrusted	-	rwx	-
Trusted/ Device Driver	-	rwx	rwx

Here the device driver will have access only to those protected objects that are identified as device driver resources.

5 Prototype Demonstrations

As part of the EXOS project, we designed and implemented a security architecture demonstration framework that illustrates the basic mechanisms behind the segment and policy managers. The demo framework has two major components: a back end (server) that runs on the SPIN machine, and a front end (client) that runs on any Java- enabled computer running the Java run-time environment.

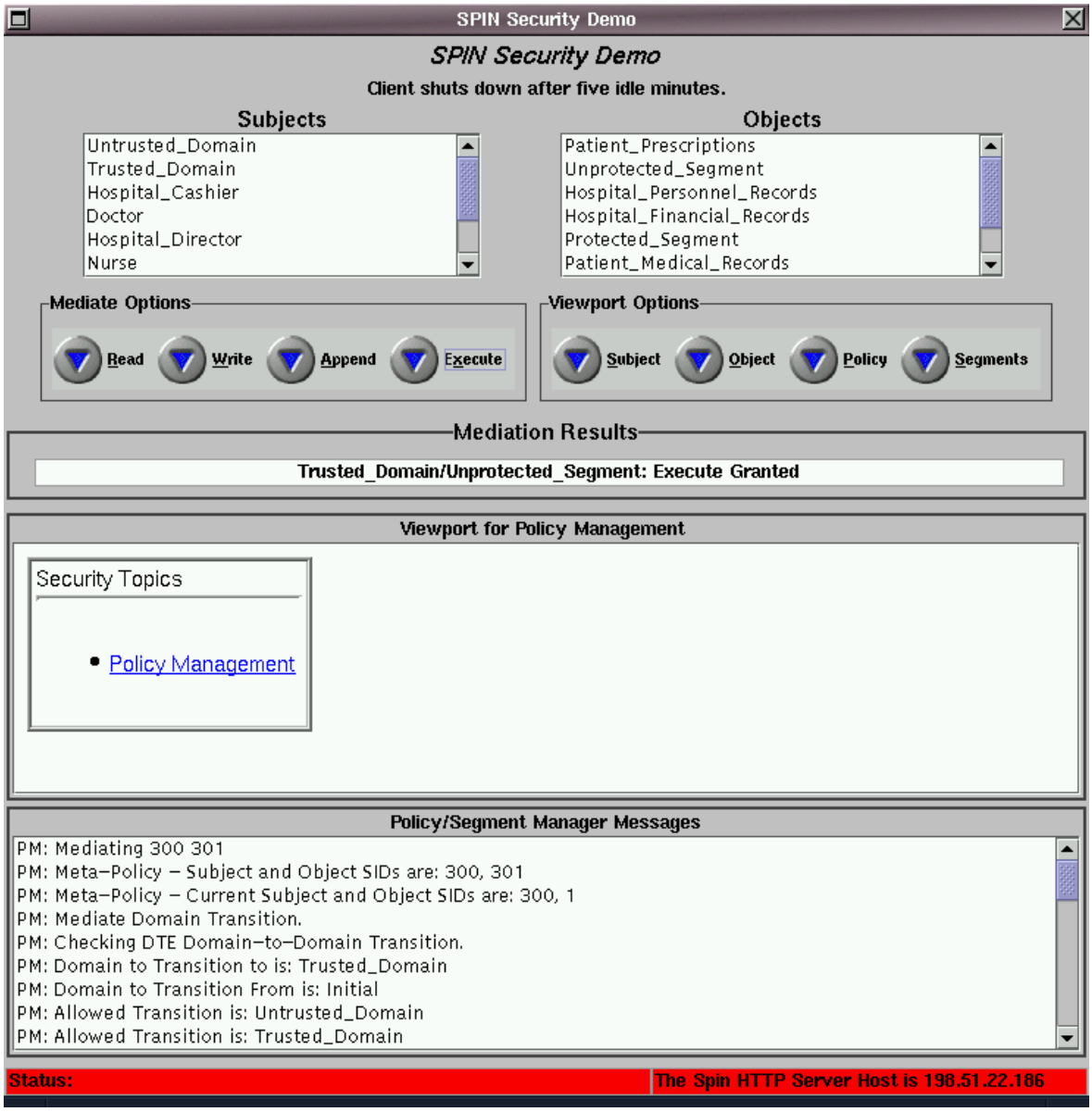
The back end of the demonstration framework consists of an HTTP server that serves up informational web pages as HTML files, which present diagnostic and illustrative information, such as the current active policy, subject and object attributes, and segment tables, to the front end. The other component of the back end is a demonstration server that creates static subjects and objects, along with their attributes, through a SPIN shell script. SPIN Policy Manager mediation for those subjects and objects is initialized by interpreting the system security policy. While the demonstration is running, the demonstration server communicates with the policy and segment managers in order to obtain the results of mediation requests.

The front end of the demonstration framework is a Java application that allows the user to view subject and object attributes, current policy information, running policy and segment manager diagnostic messages, and the results of mediation requests for subjects and objects that the user selects (see figure). The front end communicates with the back end of the demonstration framework via a stream socket for the results of mediation requests and for web page requests.

A user running the Java application can choose a subject and object (by names appearing in the applet) and a mediation option (read, write, append, or execute), and instruct the demo server to attempt the requested action. The demo server changes the SID of the currently running thread to that of the chosen subject, and attempts the requested action on the subject. If the object in question is a segment, and the subject/object pair has not been mediated before (or if the segment access cache has been flushed), the segment manager and policy manager obtain the result of the mediation (pass/fail), update the segment access cache, and retry the call. If the object in question is not a segment, the policy manager determines the result of the mediation

and retries the call. The demo server then determines whether the call passed or failed and returns that information to the front end.

The user can also see the currently active policy for the system. When the policy is requested, the back end retrieves and sends an HTML file containing the requested information. The policy HTML files are generated automatically when the policy is first read in and interpreted by the policy manager. Finally, a button exists on the Java application so that the user can see the segments available in the system. When this is requested, the back end queries the segment manager for a list of all of the segments and their address ranges, then sends that list to the front end, which displays the list in the messages window.



6 Conclusions

6.1 Lessons Learned

The results of the EXOS project lead us to conclude that important security enhancements can be made to extensible operating systems like SPIN in support of user-oriented extensibility. This security functionality can be flexibly configured to meet the security requirements generated by essentially unrestricted dynamic addition of code extensions to the kernel originating from untrustworthy user software. Just as these extensions are linked in at the procedure call level, the granularity of security checking can be tailored to security objectives based upon interception and access mediation at that same level, including both procedure visitation and parameter validation.

We based our procedure visitation and parameter validation policy enforcement on the SPIN security work being done at the University of Washington. The University of Washington security work is based on the separation of policy implementation into a policy-neutral Enforcement Manager and a replaceable Policy Manager that establishes the security policy that is active for the system. The EXOS project validated the University of Washington work by developing a very general policy manager that simultaneously supports multiple security policy models, in this case, MLS, RBAC, Chinese Wall, ACL, and DTE. A conclusion of the project is that separation of policy mediation and enforcement in extensible systems is not only desirable, but practical, and facilitates support and inclusion of policies dynamically established over the lifetime of the system. Policy-specific mediation functionality can be separated from enforcement functionality via policy-neutral security protocols such as those developed by the University of Washington.

Although useful, we consider procedure-call-interposition-based security controls to be insufficient for an extensible system such as SPIN. The potential exposure of kernel-level system service functionality to misbehaving kernel extensions resulting from single address space execution environments such as SPIN motivates the inclusion of additional security support for confinement and least privilege enforcement that further regulates control transfer and data access across security domain boundaries. These confinement and least privilege controls establish kernel-wide protection that complements the existing language-based protection mechanisms of the SPIN system.

To support confinement and least privilege, we designed and implemented a software segmentation support mechanism that we integrated into the SPIN system security architecture that could be used to provide a degree of kernel-component-to-kernel-component as well as system-to-extension isolation. We designed this mechanism to implement a form of software segments that contain and confine language-distinguished programming elements (modules). This software segment mechanism was designed to be implemented with varying degrees of dependence on hardware segmentation mechanisms for performance enhancement. In our implementation, we utilized a limited degree of hardware segmentation support for efficiency reasons to speed up the software segment lookup process, but could have extended our implementation to fully exploit hardware-based segmentation for kernel-level protection purposes had we so wished.

In considering security policy support and its realization in extensible systems, we concluded that policy specification as well as mediation support that matches established and

widely-accepted security policy model abstractions is important to the expression, implementation, and eventual evaluation of security objectives and their realization in security software. In particular, we felt that security policy should be specified by the security administrator in a form that is understandable to others with respect to the security objectives captured in a given policy. This is best accomplished if the policy can be stated in a way that is fairly intuitive. Furthermore, it is advantageous to carry the encompassing policy model context all the way through to the policy-dependent components of the security architecture, in our case, the Policy Manager. Thus, we implemented separate policy mediation modules for each policy model we decided to support. An advantage of doing this involves the ease of implementing more complicated relationships between policy components such as subject-object dominance aspect of lattice-based policy models like MLS and the temporal aspect of policy models such as Chinese-Wall. Another advantage, we believe, lies in the ease of understanding policy model implementation and relating it to correct implementation of Policy Manager mediation algorithms. This distinguishes our approach from architectures where the policy is translated to some intermediate form in which the semantics of the original policy becomes obscured by low-level specification details. To this end, our policy language as well as our Policy Manager directly supports MLS, RBAC, Chinese Wall, ACL, and DTE policy models as opposed to an intermediate policy specification language such as Adage.

The increasing emphasis on extensible system software, of which the SPIN system is only one example, suggests a requirement for security policy extensibility beyond that which might be envisioned for a more traditional operating system. As systems become more dynamic in their composition and the functionality they export to user application software, the need for safe, dynamic policy extensibility becomes more evident. It is highly likely that with an extensible operating system such as SPIN as well as other extensible systems, the introduction of middleware or new object managers would be a frequent occurrence and that such software would require extension of existing policy to support the enhanced system functionality. Our work on the EXOS project demonstrates that a useful policy extensibility strategy can be devised that supports safe kernel security policy extensibility in support of untrusted software. We investigated the issue of safe policy extensibility and identified a conservative approach to adding new policy elements to existing policy for a domain-type policy model that we feel would be especially appropriate for extensible operating systems.

As a final comment, the EXOS project was essentially a consumer of the research results of another organization, the University of Washington. In general, it is somewhat problematic to build upon someone else's research prototype (even within the same organization). Such prototypes are typically not production quality software. They have not undergone significant testing and debugging as with a commercial product. Functionality is likely to be missing or incomplete and many aspects of the architecture or its implementation will not have been stress tested or even exercised, unlike the case when software is exposed to a large body of users. In general, it is difficult to tell a priori the quality of the software and how many and what sort of problems will be encountered in working with it.

The situation tends to be compounded when the source of the software is a university-based research project. In such case, the prototype is, at least in part, a research vehicle for the investigation of topics of research interest to faculty, students, and staff. The ephemeral nature of the development team and evolving research objectives as student and faculty research needs are satisfied, particularly when it comes to student contributors, makes it especially difficult to

maintain a coherent research focus. Thus, when dealing with a university and its research products, it is important to anticipate this problem and expect a significant impact on the scope and cost of the dependant project.

6.2 Technology Transfer

Modula-3, the SPIN system implementation language is not a mainstream development language. It was developed prior to JAVA and has specific features that supported the objectives of the SPIN OS development effort and made it an attractive candidate at the time the effort was initiated. Unfortunately, Modula-3's status makes direct consumption of the resulting technology somewhat unlikely. A greater opportunity would have existed if the SPIN system had been implemented in JAVA, for example. Nevertheless, the issues investigated under the EXOS project with respect to extensible operating system security are not inconsequential, and the solutions developed for the SPIN system are potentially applicable outside of the project context.

Beyond the more pragmatic results of the project, the SPIN system provides an excellent research platform for an intrusion detection and response system prototyping effort, because of its highly flexible and adaptable nature. As part of a more comprehensive approach to ID&R, we speculate that extensions could be rolled in and out of the OS kernel to allow the system to adjust or completely modify its behavior in response to externally or internally-initiated triggering events. Specifically, SPIN system extensibility techniques might be used to support ID&R functionality that facilitated:

- Multiple simultaneous service invocation for voting and shadowing,
- Diversity of service components that support critical system functionality,
- Dynamic monitoring of service components,
- Dynamic attachment of new instrumentation and monitoring mechanisms, and
- Dynamic attachment and replacement of alternative service providers.

We believe this would yield an extremely robust ID&R platform. A *selectable* proposal (AESOP) was, in fact, submitted that advocated such work.

With respect to the general area of mobile code security research, recent papers published by researchers at the University of Utah [2][18] argue, as we believe, that the software techniques being applied to mobile code security need additional functionality. The papers argue for stronger separation than that provided by existing techniques – that mobile code execution environments need protection models similar to those employed in operating system software and that these are not only important for code protection and separation but also for resource management and communication. We believe that the software segmentation techniques developed in this effort might provide some of these additional controls.

6.3 Future Work

In terms of possible future work, we believe that it would be useful to evaluate the applicability of the SPIN security architectural techniques to other research topics. We see several areas of fruitful work. We believe that it would be useful to develop better insight into the true performance cost of SPIN security strategies on a production system. This would

primarily involve benchmarking SPIN system security functionality through a series of controlled experiments on an operational platform to determine the performance impact of the various aspects of SPIN security functional elements. We note that some limited experiments have already been conducted by the University of Washington with respect to the added overhead of null procedure calls that invoked SPIN protection functionality. However, a more detailed study is required to fully assess the added impact of realistic policy enforcement on a running SPIN operating system.

As described in this document, some attention was given to the issue of safe, dynamic policy extensibility. Specifically, we investigated DTE policy extension as it is used in our SPIN security enhancements to control access to kernel extensions through the use of software segments. We believe that it would be useful to further evaluate the utility of this policy extensibility approach to DTE, for example, with regard to how and when and with what consequences policy changes can be made at run-time as well as the applicability of the concept to other mandatory security policy models such as Role-Based Access Control, MLS, and Chinese Wall.

Perhaps the most interesting area of effort, however, involves the extension of protection techniques devised under this effort to major operating systems such as NT and Unix. Unlike the statically configured systems of the past, current systems are implementing much more dynamism in the inclusion of significant kernel-resident functional elements such as device drivers, protocol stacks, and even loadable file systems. A security domain based protection strategy can potentially provide needed security to Windows and Unix platforms to limit the capability of dynamically introduced kernel code extensions to adversely affect the remainder of the system in the event of implementation errors or even in circumstances associated with system penetration.

7 Summary

The EXOS project was tasked with adding security functionality to the SPIN operating system. In this regard it was successful in developing a policy manager that supported multiple classical security policies concurrently and demonstrating that via a demo package that could be invoked remotely. It also developed a strategy for defining and enforcing a policy on extensions based on physical properties (e.g., module association) that assigned security attributes. This technology can coexist with the basic enforcement mechanisms developed by the University of Washington's security architecture as described in [8]. The project applied a domain and type based approach to enforcing a segment's security attributes that was amenable to hardware support through standard segmentation mechanisms. It also addressed techniques for safe policy extensibility for a DTE policy model that potentially allows dynamic policy extension in support of security requirements introduced by middleware and applications that might run on the system.

8 References

[1] L. Badger, D. Sterne, D. Sherman, K.Walker, and S. Haghihat, "Practical Domain and Type Enforcement for UNIX", *Proc. of the 1995 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1995, p. 66.

[2] Godmar Back and Wilson Hsieh, "Drawing the Red Line in JAVA", *Proc. of the 7th IEEE Workshop on Hot Topics in Operating Systems*, Rio Rico, AZ, March 1999.

[3] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiucyzynski, Craig Chambers, and Susan Eggers, "Extensibility, Safety and Performance in the SPIN Operating System", *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, CO, December 1995, pp. 267-284.

[4] W.E. Boebert and R.Y. Kain, "A Practical Alternative to Hierarchical Integrity Policies", *Proc. of the 8th National Computer Security Conference*, 1985, pp. 18-27.

[5] Department of Defense National Computer Security Center, *DoD Trusted Computer System Evaluation Criteria*, 1985 (Orange Book).

[6] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole Jr., "Exokernel: An Operating System achitecture for Application-level Resource Management", *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, Copper Mountain Resort, CO, December 1995, pp. 251-266.

[7] Paul Green, "Multics Virtual mmory - Tutorial and Reflections", Technical Report available on the Internet at: http://www.cs.berkeley.edu/~brewer/cs262/Green.ps.

[8] Robert Grimm and Brain N. Bershad, "Access Control in Extensible Systems", University of Washington Department of Computer Science and Engineering Technical Report UW-CSE-97-11-01, November 1997.

[9] Robert Grimm and Brain N. Bershad, "Providing Policy-neutral and Transparent Access Control in Extensible Systems", University of Washington Department of Computer Science and Engineering Technical Report UW-CSE-98-02-02, February 1998.

[10] Elliott I. Organick, *The Multics System: An Examination of Its Structure*, MIT Press, 1980.

[11] Przemyslaw Pardyak and Brian N. Bershad, "Dynamic Binding for an Extensible System", *Proc. of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, October 1996, pp. 201-212.

[12] Ravi S. Sandhu, "A Lattice Interpretation of the Chinese Wall Policy", *Proc. of the 15th NIST-NCSC National Computer Security Conference*, Baltimore, MD, October 1992, p. 221.

[13] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman, "Role-based Access Control: A Multi-dimensional View", *Proc. of the Tenth Annual Computer Security Applications Conference*, Orlando, FL, December 1994, pp. 54-62.

[14] Ravi S. Sandhu, “Role-based Access Control”, Laboratory for Information Security Technology Technical Report, September 1997.

[15] Stefan Savage and Brian N. Bershad, “Issues in the Design of an Extensible Operating System”, *Proc. of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, November 1994.

[16] O.S. Saydjari, J. Beckman, and J. Leaman, “LOCKing Computers Securely”, *Proc. of the 10th DoD/NBS Computer Security Conference*, Gaithersburg, MD, September 1987, p. 129.

[17] K. Walker, D. Sterne, L. Badger, K.A. Oostendorp, M.J. Petkac, and D.L. Sherman, “Confining Root Programs with Domain and Type Enforcement”, *Proc. of the 1996 USENIX UNIX Security Symposium*, San Jose, CA, July 1996.

[18] Patrick Tullmann and Jay Lepreau, “Nested Java Processes: OS Structure for Mobile Code”, *Proc. of the 8th ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[19] Mary Ellen Zurko and Richard T. Simon, “User-centered Security”, *Proc. of New Security Paradigms Workshop*, September 1996.