Security Mechanisms and Policy for Mandatory Access Control in Computer Systems

By

Glenn Daniel Wurster

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

Carleton University
Ottawa, Ontario, Canada

Abstract

Computer security measures, policies and mechanisms generally fail if they are not understood and accepted by all parties involved. To be understood, many security mechanisms currently proposed require security expertise by multiple parties, including application developers and end-users. Unfortunately, both groups often lack such knowledge, typically using computers for tasks in which security is viewed at best as a tertiary goal. The challenge, therefore, is to develop security measures understood and accepted by non-experts.

We pursue measures which require little or no user expertise, to facilitate broad deployment among non-technical user bases. By reducing the requirement that end-users self-police applications, we reduce the chance of policy enforcement errors causing security exposures. The security measures discussed are also straightforward and intended to avoid reliance on security expertise among application developers. For example, restrictions imposed by an application's target run-time environment essentially remove development choices (thus removing dependence on the developer to make proper security choices). We pursue measures designed to be suitable for deployment to large segments of the development community, to reduce the knowledge and adoption barriers that may otherwise arise. The security measures we propose provide protection by significantly restricting the operations that an application is allowed to perform.

To address issues related to malicious sites and dangerous interactions between sites, we discuss the joint work SOMA, a browser extension. SOMA enforces a security policy that limits interaction between web sites to those that are pre-approved by one or (optionally) both sites involved in any interaction. SOMA can be incrementally deployed for incremental benefits, and selectively deployed to those sites for which tighter control over content sub-syndication is acceptable. To address rootkits and malware affecting the installation and integrity of binaries, we present three policies; `configd`, bin-locking, and increased kernel protection. For each approach, we discuss the architecture, implementation and support required. These ideas are suitable for many types of end-user machines, including those running Linux and Windows. They do not require any centralized infrastructure. We discuss approaches which do not depend on either software developers or users to properly address software security.

Acknowledgements

First and foremost, I want to acknowledge my wife, Heather, for her love and support during the process of completing my doctorate. I also want to thank my parents for their many years of support and parental guidance.

My supervisor, Paul C. van Oorschot was invaluable during the process of performing both the research as well as writing the resulting papers (and this thesis). His prompt responses and detailed critiques resulted in a much better end product. His wealth of security related knowledge and insights were appreciated. I'm thankful for his ability to step back, pulling out the core elements of an idea and helping me express them clearly.

To those who reviewed this thesis, your help was much appreciated. Terri Oda, Douglas Paul and Heather Wurster, your comments were much appreciated and helped make the final product that much better.

Thanks to many of the colleagues in the Carleton Computer Security Lab who provided feedback during the course of the Ph.D. Their critiques and comments helped fine-tune ideas. Conversations with Mohammad Mannan, Julie Thorpe, David Whyte, Terri Oda, David Barrera, and Carson Brown helped solidify ideas into workable concepts.

For the anonymous conference referees which reviewed the papers submitted to conferences, I wish to thank you for your constructive feedback. One of the most important avenues for feedback is through peer reviewed submissions – your constructive comments were appreciated. For those on the thesis defence committee, I likewise thank you for your feedback.

I finally want to acknowledge funding received through a NSERC PGS D scholarship, as well as funding from NSERC ISSNet and Carleton University.

Contents

List of Tables

List of Figures

1 Introduction

In current computing environments, applications written by many different authors all co-exist, sharing physical resources which include the network and file-system. Protecting these applications against attacks (both network and file-system based) while allowing them to share physical resources is not a simple task, especially since some of the applications may be malicious. The sharing of physical resources between many different applications brings with it many security issues which need to be dealt with. While there are many people involved with computers on a daily basis, only a small number of them have the skills required to protect applications against attack. Two groups of individuals who have often been tasked with more security responsibilities than their abilities warrant are users and application developers.

As computers have become more widespread, users of all different skill levels with a variety of skill sets have started buying and interacting with them on a daily basis. The number of application developers has similarly exploded in recent years, with many of them also having different skill levels and specializing in specific areas. Developers can focus on mobile applications, data mining, web page development, game design, image processing or any number of other areas related to software development. Expecting every developer to be an expert in computer security is ill-fated. The developer has many goals and pressures which influence how they choose to spend their time when developing software. With security a tertiary goal (at best), we cannot depend on application developers to develop secure software. We believe that relying excessively on either users or application developers for security is ill-advised. As computers become more accessible to both developers and users, the level of knowledge which can be assumed has decreased.

The increased accessibility of computers has resulted in a situation where both users and application developers are using computers while not being familiar with all the details of the system. We believe this is a strength of modern computers, one which allows much broader deployment. With this strength, however, comes a caveat; we can no longer assume in-depth knowledge by either in properly securing a system. We focus in this thesis on several methods for better protecting applications on a system while not depending on either group to properly understand, implement, or respond to complex security mechanisms. We do this by restricting the damage an application can do when abused or compromised to limit the consequences when security is breached (or, in the case of malware, we restrict the activities it is allowed to perform at all). Our restrictions do not depend on a security-aware user for policy enforcement, since the mechanisms we introduce are designed for computers owned and operated by non-expert users. Our aim is to restrict the damage a malicious application can do (either by design, or when the security of an application is compromised through a vulnerability); therefore, it is only natural that many of the protection mechanisms we discuss are designed to prevent applications from interfering with each other.

1.1 Segregation of Applications

The specific protection mechanisms we introduce can be classified based on the environment the associated application is designed to run in. In this thesis, we concentrate on developing protection mechanisms for two such environments, applications designed for desktops and those designed to be deployed as websites.

1.1.1 On the Desktop

In current computing environments, applications written by many different authors all coexist on disk, being installed at various times. Each application normally includes a number of program binaries and libraries, along with some associated data and configuration files. While the installation of a new application will normally not overwrite files previously installed by another application, permission is commonly granted to modify all binaries (although this is not generally understood by all end-users). Indeed, this raises a problem: *Any application or application installer running with sufficient privileges can modify any other application on disk.* Applications installers are routinely given these privileges during software upgrade or install (e.g. almost all installers run as administrator or root, giving complete access to the system). Some applications even run with administrator privileges during normal operation despite the best efforts and countless recommendations against this practice over the years. Commonly running applications (including their installers) as administrator leads to a situation in which a single application can modify any other application's files on disk. Normally, applications do not use (or abuse) this privilege. Malware, however, does not traditionally respect the customs of normal software, and uses the ability to modify other binaries as a convenient installation vector. Already in 1986, the Virdem virus [154] was infecting executables in order to spread itself. More recently, rootkits [51] have used binary file modification in an attempt to hide.

One part of this thesis pertains to protecting an application's files on disk. We discuss two different protection mechanisms: Bin-Locking, which is designed to limit modifications to binaries on disk, and `configd`, which is designed to limit modifications by an application to other application's file-system objects.

1.1.2 As a Website

Current web pages are more than collections of static information: they are a combination of code and data often provided by multiple sources, being assembled or run by the browser. The browser itself provides a very powerful environment for communicating with arbitrary web servers and running arbitrary web-based applications. External content fetched by a browser may be untrusted, untrustworthy, or even malicious. Such malicious content can initiate drive-by downloads [135], misuse a user's credentials [63], or even cause distributed denial-of-service attacks [99].

A common thread in misuse of the functionality provided by a browser is that the browser must communicate with web servers which would generally not be contacted during the normal execution of the web-based application. Those servers may be controlled by an attacker, may be victims, or may be unwitting participants. Whatever the case, information should not be flowing between the user's browser and these sites.

A second part of this thesis addresses restricting the exposure of web-based applications in an effort to reduce the effect of some of the most common web-based attacks. We detail and discuss the *Same Origin Mutual Approval* (SOMA) approach, which requires the browser to verify that both the site operator of the page and the third party content provider approve of the inclusion before any communication is allowed (including adding anything to a page).

1.2 Thesis Statement

From the above discussion, the doctrine which motivates our work is summarized by the following assumptions:

A1. The current approach of relying on the application developer to properly implement security policies that protect a system against attack is inappropriate, given that many developers are not experts in security (and indeed, some developers write malicious applications intentionally). Similarly, relying on an end-user to police security policies is unwise, given that many end-users are not educated in security.

A2. Applications written by different developers co-exist, sharing resources. While we focus on sharing of the file-system and Internet, the assertion holds true for other physical resources as well.

A3. Applications rely on an environment provided by some external third-party. In the case of desktop applications, this is the OS vendor. Web applications similarly rely on both the web server they run on, as well as the browsers.

A4. Those creating an application environment can be relied upon to properly implement security mechanisms designed to protect applications operating in the environment from interfering with each other. We do not assume that those creating the application environment will be able to design new security mechanisms (the contribution of this thesis is in designing new security mechanisms specific to two environments).

Given the above assumptions, we hypothesize that there are approaches that can be taken for protecting applications that do not require end-users or developers to be security experts, and result in better overall application security. Given that both the system owner and application developer may not be relied upon to adequately protect applications, can appropriate mandatory access control mechanisms be developed to better protect applications? Our objective is to pursue this question, and if possible, design several mechanisms that result in better overall application security but require little end-user or developer security expertise. In pursuing this thesis, the directions the research took involved finding such mechanisms for both desktop applications, as well as web applications. A second goal of our work is to draw more attention to an under-explored subset of mandatory access control policies, which we call guardian and define in Section 2.3.3.

1.3 Main Contributions

In this thesis, we introduce four access control mechanisms, which impose additional limits on application software. These mechanisms (and the policy they enforce) are designed to be easy for an application developer to understand and work within. They are also designed to require minimal user involvement. The specific mechanisms we introduce are:

1. **Limiting Privileged Processor Permission** - We consider in Chapter 4 a policy for restricting the ability to run arbitrary code with privileged processor permission. We pull together pre-existing protection rules and introduce new protection rules designed to separate root from kernel-level privileged processor control on a desktop system. In doing so, we provide a basis for positively answering the thesis question. The protection mechanism resulting from combining the rules does not assume any additional security knowledge by end-users, satisfying the constraints of our thesis question. In separating user from kernel level processor control, we also protect uneducated users from operations that can result in file-system data loss. We implement the protection mechanism on a prototype system, evaluating both its performance and ability to protect against current rootkit malware.

2. **Bin-Locking** - We consider in Chapter 5 the problem of operating system and application binaries on disk being modified by malware. We present a new file-system protection mechanism designed to protect the replacement and modification of binaries on disk while still allowing authorized upgrades. We use a combination of digital signatures and kernel modifications to restrict replacement without requiring any centralized public key infrastructure. Such an approach affirmatively answers the the thesis question. To explore the viability of our approach, we implement a prototype in Linux, test it against various rootkits, and use it for everyday activities. The system is capable of protecting against rootkits currently available while incurring minimal overhead costs. We do not protect configuration files, instead focusing on protecting binaries the user does not modify. `Configd` addresses the protection of configuration files.

3. **Configd** - In Chapter 6, we address the problem of restricting root's ability to change arbitrary files on disk in order to prevent abuse on most current desktop operating systems. The approach involves first recognizing and then separating the ability to configure a system from the ability to use the system to perform tasks. The permission to modify configuration of the system is then further subdivided in order to restrict applications

from modifying the file-system objects of other applications. We explore the division of root's current ability to change arbitrary files on disk and discuss a prototype that proves the viability of the approach. The novelty in the approach comes from being able to protect, on a desktop used by non-experts, an applications file-system objects on disk. The approach affirmatively answers the thesis question.

4. (joint work)[1] **SOMA** - In addition to the main contributions of this thesis, we also discuss and expand on SOMA. Unrestricted information flows are a key security weakness of current web design. The SOMA approach as discussed in Chapter 3 fits well with our thesis goal of better protect applications from attack, without relying on expertise on the part of either the system owner or application developer. By requiring site operators to specify approved external domains for sending or receiving information, and by requiring those external domains to also approve interactions, we prevent page content from being retrieved from malicious servers and sensitive information from being communicated to an attacker. SOMA is compatible with current web applications and is incrementally deployable, providing immediate benefits for clients and servers that implement it. SOMA does not depend on the web application developer or browser user for proper enforcement, satisfying the constraints of the thesis question.

1.4 Related Publications

Many parts and ideas contained in this thesis have been peer-reviewed. These publications are listed below in chronological order.

1. G. Wurster, P. C. van Oorschot. *Self-Signed Executables: Restricting Replacement of Program Binaries by Malware*. In Proc. 2007 Workshop on Hot Topics in Security (HotSec), August 2007 [189].

    - Chapter 5 details the contents of this paper.

2. G. Wurster, P. C. van Oorschot. *The Developer is the Enemy*. In Proc. 2008 Workshop on New Security Paradigms, September 2008. pp 89-97 [190].

[1]The SOMA work appeared first in a paper [126] published with another Ph.D. student co-author. The author of the present dissertation contributed the idea of isolating applications. Terri Oda contributed domain-specific knowledge required to make the approach feasible when applied to web applications.

- This workshop paper explored the subject of not trusting developers to always make proper security choices. While we do not incorporate it in its entirety within the thesis, at a high level the thesis draws on concepts introduced in this work. In this thesis, we treat the developer as being unable to properly make security related decisions.

3. T. Oda, G. Wurster, P.C. van Oorschot, A. Somayaji. *SOMA: Mutual Approval for Included Content in Web Pages*. In Proc. 15th ACM Conference on Computer and Communications Security, October 2008. pp 89-98 [126].

   - Chapter 3 details and expands on the contents of this paper, which is joint work.

4. G. Wurster, P. C. van Oorschot. *System Configuration as a Privilege*. In USENIX 2009 Workshop on Hot Topics in Security (HotSec), August 2009 [191].

   - Chapter 6 details the contents of this paper.

In addition, this thesis presents additional work not yet published in a peer reviewed journal or conference.

1. G. Wurster, P. C. van Oorschot. *A Control Point for Reducing Root Abuse of File-System Privilege*. Under submission to conference.

   - Chapter 6 details the contents of this paper.

2. G. Wurster, P. C. van Oorschot. *Towards Reducing Unauthorized Modification of Binary Files*. Technical Report TR-09-07, Carleton University, September 2009 [192]. Under submission for journal publication.

   - Chapter 4 and 5 detail the contents of this paper.

1.5 Organization

In Chapter 2, we provide background on mandatory access control policies, including several which have already been deployed. We concentrate on how each of the MAC policies is enforced, focusing on the demands placed on both the user and application developer. In Chapter 3, we discuss SOMA, an approach for better isolation of applications which have been developed for the web (i.e., the application relies on both a web server and web browser communicating over the same network). Chapter 4 introduces additional enforcement mechanisms which limit any applications ability to gain privileged processor control. Such protections are used in Chapter 5, which introduces an approach for limiting updates to files based on whether the update can be verified based on data contained in the already-installed file. Chapter 6 uses the mechanisms discussed in chapter 4 and 5 to protect file-system objects belonging to an application from being modified by other applications on the system. We provide a summary of the approaches in Chapter 7, revisiting the thesis hypothesis and questions.

2 Overview

In this chapter, we provide an overview of access control policies and related mechanisms as they relate to this thesis. We start our discussion by examining the role of a system administrator in securing the environment they are placed in charge of. We then discuss the various types of access control policies, along with how they can be enforced. We also present several access control mechanisms that fall into the same category as those presented in this thesis.

2.1 A System Administrator Analogy

The job of a system administrator is often conflicting. They must provide support to the users in a particular environment while keeping that environment secure. This involves ensuring that all tools users require to get their job done are available while at the same time not allowing the users to customize their systems to the point that they become vulnerable to attack [26]. Furthermore, administrators usually prefer to keep the environment identical amongst all of the computers they are maintaining to reduce maintenance overhead. The good system administrator will leave the computers in such a state that users do not feel restricted in how they perform their tasks, but at the same time maintain control over the system in order to combat malware. The job is made harder by the fact that many users are not security experts in-themselves, and consequently may do things that undermine the security of the system.

This situation, the continual balance between the goals of the user and goals of system administrator, can be parallelled in the software development world. In such an environment, users become the software developers and system administrators are those that create the environment used by developers. This can either be the software development environment or the run time environment of the developed software. Stray too far toward giving the developers total control of the systems they are developing for and the security of the system will suffer as a result. Stray too far toward limiting developers, and they will likely flee from the environment, preferring instead something potentially less secure but more usable. Most software developers are not security experts and so, like standard users, may do things that will undermine the security of the resulting system. In reality, both users and software developers are attempting to get their job done, and security is often not a primary task [19].

If we examine the security mechanisms that have been applied to users versus those that have been applied to developers, we see a large discrepancy. While we have been attempting to enforce many security policies on a user (e.g., password strength), many of the security policies have not been enforced upon developers (e.g., buffer overflow detection), instead being offered as an option developers can choose to use. Developers, not surprisingly, are unlikely to choose the security option unless some other external influence also exists to make the choice to change their behaviour/routine more appealing. How do we increase the security of our system? We convince developers to exchange current unsafe approaches for safer ones.

2.2 Promoting Change

It is commonly said that we are creatures of habit. We have preferred ways of doing things and breaking a habit can be an arduous task. Individuals also prefer to stick with what they know. As a developer, we prefer to use tools, technologies, and approaches with which we are familiar [97]. This is especially the case when we are placed under stress (e.g., not having time to learn new approaches). For this reason, the thesis concentrates on security mechanisms that are simple for both the application developer and the end user. We pursue measures that require little or no user expertise, to facilitate broad deployment among non-technical user bases. By reducing the requirement that end-users self-police applications, we reduce the chance of policy enforcement errors causing security exposures. By reducing required developer knowledge and hence adoption barriers, we facilitate use by large segments of the development community. Approaches either fly or die based on whether they are picked up by the general community, so making deployment as easy as possible is critical.

2.3 Access Control Types

We now discuss necessary background related to security mechanisms and the policies they enforce. There are several different types of access control, differentiated by who is in control of permissions related to the elements being protected by an access control mechanism. The taxonomy is illustrated in Figure 2.1.

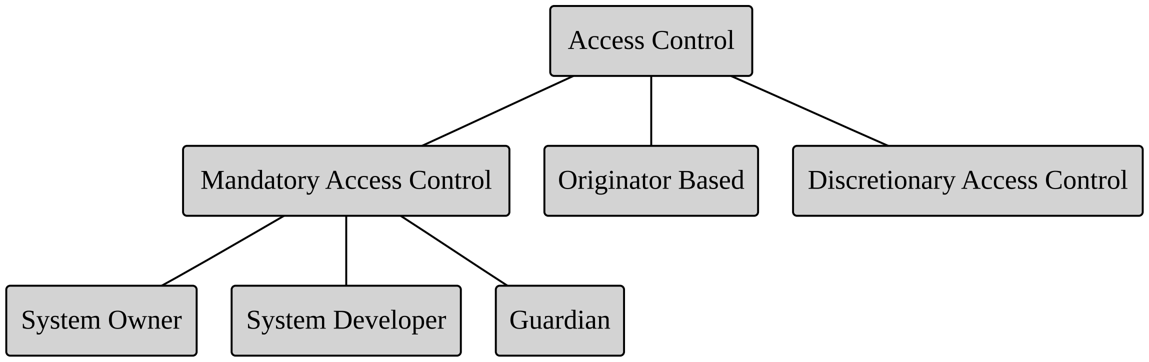


Figure 2.1. A taxonomy of types of access control. We build on the generally accepted taxonomy of access control [24] by further subdividing the class of mandatory access controls based on who is responsible for setting policy.

2.3.1 Discretionary Access Control

Under a discretionary access control method, an individual who owns an object can either allow or deny access by others to the object. One example of this is the typical POSIX access controls on Unix, where the owner of a file is allowed to set read, write, and execute access for other members of the group and everyone else.

2.3.2 Originator Controlled Access Control

Under originator controlled access control, the ability to access an object is controlled by the creator of an object [24]. A common example of such an approach is digital rights management in digital media, where the creator attempts to retain control over access to the work after it is distributed to the buyer [101]. Bin-locking (Chapter 5) comes close to being such an approach, because the creator is capable of restricting updates based on the selection of keys embedded into the file. In reality, however, bin-locking is best thought of as a hybrid between mandatory and originator controlled access control.

2.3.3 Mandatory Access Control

Under a mandatory access control, access policy for an object is set by an individual other than the owner or creator of the object. Such a policy is enforced through one or more security mechanisms that exist on the system. The user, even if they own the object, cannot change the mandatory access control policy for that object. Processes that run for the user cannot modify the policy either. Typically, policy is set by another individual (or same individual assuming a different role). In this thesis, we sub-divide the field of mandatory access control based on who is responsible for setting policy.

Policy set by System Owner

In this environment, the mandatory access control policy is controlled by the owner of a system (often the system administrator). If the equipment is part of a company, then either the IT department or management is typically responsible for setting policy. In home environments, the owner and system administrator are typically one and the same. It is assumed that the system administrator is capable of setting the correct mandatory access control policy and properly implementing the underlying policy mechanisms required to enforce the policy.

Policy set by System Developer

In this environment, the mandatory access control policy (and deployment of related mechanisms) is set by the developer of the software or hardware. One reason these policies may be enforced by the developer is to limit damage to either the hardware or software (e.g., software's ability to control the refresh rate in CRTs is limited to prevent damage due to incorrect refresh rates [124]). Other policies may be imposed by the developer to prevent software vulnerabilities such as buffer overflows (e.g., limiting the size of a string which can be processed by the software).

Policy set by a Guardian

While the system administrator or developer in charge of the mandatory access control policy may have sufficient knowledge to properly set policy, there is also a chance they will not. This is, in fact, a common criticism against SELinux

[123]. For this reason, we introduce a third category for how mandatory access control policies can be set: by a *guardian*. In this enforcement approach, the policy is set by someone who is knowledgeable in setting appropriate policy in order to secure a system (developers and system administrators can be included in this category as long as they are capable of properly setting the security policy and related enforcement mechanisms). They are designed to protect individuals who are less aware and to be administered by those who are more aware of threats inherent in the particular system.

An example of guardian enforced mandatory access control are parental controls embedded into many media players and video game consoles. Parental controls do not place control of content in the hands of content creators or developers, nor directly into the hands of the system owner (who may be a child in the case of a video game unit). Rather, they are designed to put control into the hands of the parents, who are capable of making informed decisions about the risks and benefits in a particular environment. Parents are assumed to be sufficiently aware of the dangers of age-inappropriate material to make informed decisions about the content to restrict.

SELinux can be shifted to a guardian enforcement approach by having a central repository of policies maintained by experts which can be drawn from in protecting a particular system. Recent advancements in SELinux take this approach [118]

2.4 Information Policy Taxonomy

The various policies for controlling the flow of information are illustrated in Figure 2.2 [24]. Policies generally fall into one of three categories: focusing on the integrity of information, the secrecy of information, or some combination of the two. For those policies focused on integrity, the goal is to prevent modifications which reduce the integrity of the information (i.e., modifications which make the information incorrect or unreliable). For security-based policies, the goal is to prevent the information being disclosed to those not authorized to view it. Hybrid policies combine aspects of both integrity and secrecy based information policies. The Chinese wall policy includes elements from both the integrity and and confidentiality policies. It helps prevent against conflict of interest in the financial world [27].

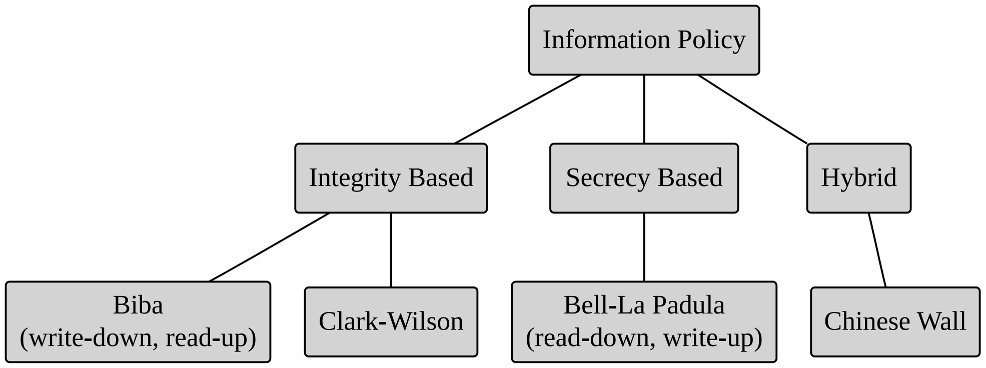


Figure 2.2. A taxonomy of access control policies [24].

2.4.1 Bell-LaPadula Security Model

In the Bell-LaPadula model [20, 21], a set of labels are created that define the sensitivity level of the information associated with that particular label. The typical example of such an ordering is 1) unclassified, 2) confidential, 3) secret, and 4) top secret. Any piece of information labelled top secret would be strictly more sensitive than information labelled confidential. In such a system, each end-user is also assigned a clearance, indicating what level of information they are allowed to access. In our example, we consider an individual Heather who is granted secret permission.

Heather is allowed to read all information classified as secret as well as information classified as both confidential and unclassified (i.e., she can read down). Furthermore, Heather is also allowed to write to information with higher classifications (i.e., she can write up). In allowing Heather to write up, she can communicate information to those who may have higher clearance levels, but is prevented from leaking information to those who do not have as high a clearance level. In the Bell-LaPadula system, content assigned to a specific security level can never be leaked to a lower security level.

The simplistic model of read down, write up is not sufficient in protecting higher level information from being corrupted, because anyone is capable of modifying top secret information, even though they will not be able to view it. For this reason, additional discretionary restrictions are placed on the ability to read and write to content. The Bell-LaPadula Model therefore combines both discretionary and mandatory access control policies.

2.4.2 Biba Integrity Model

In the Biba integrity model [23], the system is designed to preserve the integrity of objects on the system. Similar to the Bell-LaPadula model, each piece of information is assigned a security level, and principals in the system are assigned clearances. In contrast to Bell-LaPadula, however, the focus is on integrity as opposed to secrecy. In Bell-LaPadula, Heather was allowed to write to content at a higher secrecy level and read from content at a lower secrecy level. In Biba, Heather would be allowed to do the opposite – read content at a higher integrity level and only write to content at a lower integrity level. In this way, content with high integrity can only be modified by principals who have permission to write to these objects, even though it can be read by everyone. The restrictions on read are enforced to prevent someone with permission to modify high integrity data from incorporating low integrity data into high integrity data.

The write down aspect of Biba is similar to the POSIX model of root being able to write to all files on disk and users being only able to read the files. A difference, however, is that root is also allowed to read all files on disk, an action not allowed by the Biba integrity model.

As with the Bell-LaPadula model, it is possible to add additional discretionary access control policies on top of the base Biba model to further restrict operations on the system (e.g., you may not want everyone to be able to read all information designated as higher integrity). As long as the mandatory access controls of Biba are not broken, system integrity is assured.

2.4.3 Clark-Wilson Integrity Model

The Clark-Wilson integrity model [35] concentrates on the transactions (or *transaction procedures* - TPs) that can be performed on an object. In their model, the system state must be consistent before and after each transaction (each transaction is composed of one or more operations that, if run individually, could leave the system in an inconsistent state). In this model, the elements that must stay consistent are considered *constrained data items* (CDIs). After the CDIs are verified as initially being consistent by an *integrity verification procedure* (IVP), one can be guaranteed that any allowable transaction will leave the CDIs in a state which is also verifiable using the IVP. In order to maintain integrity, only verified TPs are allowed to run, and each verified TP can only be run by a user with sufficient permission (i.e., each user is specified as having a set of allowable TPs that they can run). Additional requirements are imposed to enforce separation of duty, user authentication, and sufficient logging of activities.

2.5 Related Access Control Mechanisms

In this section, we discuss several access control mechanisms related to the work in this thesis.

2.5.1 SELinux

Given that most of the security mechanisms discussed in this thesis are for Linux, we would be remiss if we did not discuss SELinux [102, 107]. SELinux provides both an enforcement mechanism and an associated specification language for deploying a mandatory access control policy [118]. The specification language is based on three elements; subjects, objects, and actions. Subjects are the actors (running processes) on the system. They are allowed to perform certain actions on specific objects. The enforcement mechanism used in SELinux relies on being able to determine whether the subject is authorized to perform a specific action on a object based on a table look-up. Such an approach is capable of being used for type enforcement, role based access control (RBAC), multi-level security (MLS) and additional discretionary access control (DAC) [118].

The base NSA SELinux system falls into the category of mandatory access control policies that are specified by the system administrator. With the introduction of policies created by experts and distributed by a central authority [118], the implementation switches to a guardian style of access control enforcement. The complexity of SELinux has greatly limited its deployability to date, resulting in the system being used in very limited contexts and with expert users [123]. The approach of having guardians (instead of end-users) create SELinux security policies has seen broader deployment.

While this thesis introduces new protection mechanisms, SELinux has a fixed protection policy (an access control matrix) and allows the labelling of subjects and objects to vary. The difference is subtle but important. As flexible as SELinux security policy is, some protection mechanisms discussed in this thesis are not easily described using the specification language of SELinux, including SOMA, bin-locking, and `configd`. SOMA exists in a different application environment, bin-locking allows the creator of the application binary to allow or deny replacement, and `configd` applies to installers, which are not likely to have a SELinux policy present on the system (or if they do have policy, it is very accommodating).

2.5.2 JavaScript Same Origin Policy

The JavaScript same origin policy is designed to limit the ability of JavaScript to read from and write to content retrieved from a different origin. It is another example of mandatory access control policy being set by a guardian. Policy and mechanism are both set by the browser manufacturers, not web developers or users. We discuss the JavaScript same origin policy in detail in Section 3.1.1.

In contrast to SELinux, the JavaScript same origin policy is simple. Also in contrast to SELinux, the same origin policy enjoys widespread deployment and use amongst non-technical users (most of whom don't even know it is in use). Its relatively simple fixed policy and lack of user involvement in enforcement are some of its greatest strengths.

2.5.3 The Apple Application Marketplace

The procedure for loading an application on many of Apple's devices (including the iPod, iPhone, and iPad) currently involves having the application developer submit their application to Apple for verification before it is made available for download from the application market (a similar approach was previously used in Symbian phones). Users may only download and install applications from the market. The Apple application marketplace is another example of a mandatory access control policy being set by a guardian, where security decisions are made by Apple as opposed to either the developer or the end user. While most other approaches assume a user with physical possession of the device (and sufficient know-how) can ultimately modify the mandatory access control mechanism, Apple attempts to prevent those in physical possession of the device from disabling or otherwise modifying the protection mechanism.

The approach taken by Apple falls into the class of mandatory access control policy being set by a guardian, similar to those presented in this thesis. In this thesis, however, we choose not to attempt to protect against physical attacks, leaving open the possibility that the user in physical possession of the device (and sufficient technical know-how) may disable the mandatory access control.

2.6 Characteristics of Policies Presented

The policies and related security mechanisms discussed in this thesis fall into the category of mandatory access controls that are set by a guardian (recall Figure 2.1 on page 11), being controlled in such a way that we hope decisions are made by those educated and capable of choosing appropriately. In addition, the security mechanisms discussed in this thesis have two additional attributes:

1. They involve little or no user involvement for their enforcement.
2. They have a low adoption barrier for developers who must abide by them in their software applications.

We further define the guardian set mandatory access control policies discussed in this thesis by incorporating desirable aspects from parenting – tenderness and firmness. In rearing good computer users and developers, we follow the same approach of being tender (by providing an environment which is constructive rather than adversarial) and being firm about the boundaries that cannot be crossed (to provide greater protection against security risks). For users especially, having the policy be set by a guardian helps the user avoid harm even though they may not fully appreciate the risks (similar to how parents protect children from unknown dangers). In such a system, applications (and the developers who create them) are likewise not given full trust. Firm limits which prevent various dangerous activities are imposed in the interest of protecting users, applications, and the environment they run in. In another analogy to parenting, the protection mechanisms should not be overly restrictive or those affected may rebel.

The security mechanisms discussed in this thesis are designed to be both applied to individuals less aware of and administered by those more aware of dangers inherent in the particular system: for desktop applications, these include the OS designers, and for web applications, the browser vendors and web server administrators. Those developing content must abide by the policy set by the guardian (the OS designer, browser vendor, or server administrator) in order to be accepted on the system.

2.6.1 Execute Disable as an Example Mechanism

Execute disable blocks a processor from running binary code from a page in memory unless the page has been marked as executable in the page table [69]. This feature is implemented on most modern processors in an effort to protect against code injection buffer overflow attacks [39]. The execute disable bit is implemented in hardware, enforced by the processor, activated by the operating system, enforced in all applications, and virtually hidden from users.

The introduction of execute disable as a method for combating code injection vulnerabilities is also an example of a new mandatory access control mechanism being introduced by a guardian into an already established software ecosystem. It was introduced in the Windows environment with the release of service pack 2 for Windows XP [112]. While initially the approach was an opt-in one, it is enabled for all 64bit applications [66]. Linux has also introduced execute disable functionality. The successful introduction of execute disable gives evidence that it is possible to add new mandatory access controls into an already deployed system (although some introductions require changes to existing apps).

The ability to both execute and write to data on the same page is a feature that is not commonly used by most developers. Most compilers automatically store data on different pages of memory than code. Applications are generally not negatively affected by the introduction of execute disable (e.g., Arjan van de Ven indicated no applications were broken by the enabling of execute disable in Fedora Core 1 [171]).

The execute disable bit does have its limitations, namely, it is not able to prevent data from being executed when the data is located on the same page as code. Also, it is not able to prevent against the modification of data [151], or return-into-libc attacks [150]. The simplicity of the approach, however, has led to its widespread acceptance in spite of the deficiencies. While many other solutions to the problem of code injection attacks have been proposed (e.g., [14, 82, 140]), execute disable continues to be the most popular method because of its simplicity and broad support.

The introduction of execute disable into modern desktop operating systems and its enforcement on all applications running on the system are the properties we are seeking for the new access control policies introduced in this thesis. Execute disable increased the security of the overall system because it was widely deployed, did not depend on each developer to selectively enable the feature, and did not depend on the user to understand and administer the protection mechanism. It is therefore a good example of the types of parental style guardian enforced mandatory access control mechanisms which are discussed in this thesis.

Execute disable does not prevent all buffer overflow attacks that can occur, but the combination of it with other approaches, such as address space layout randomization (ASLR [151, 39]), has made exploiting a buffer overflow much more difficult for an attacker. ASLR, like the execute disable bit, is controlled by the operating system and enforced on applications.

2.7 Summary

The mechanisms introduced in this thesis are designed to be set by a guardian. They are designed to be enforced by those designing the environment that applications will run in. They are also designed to be applicable to environments where the user may not be a security expert. As evidence that guardian enforced mandatory access control mechanisms can be deployed into pre-existing environments, we discussed execute disable in Section 2.6.1.

3 Same Origin Mutual Approval Policy

In this chapter, we present and build on the *Same Origin Mutual Approval* policy (SOMA), joint work developed to restrict the fetching of web content. Following in the style of parental style guardian mandatory access control policies, the approach does not rely on user interaction and imposes additional restrictions on all developers creating web applications. SOMA is designed to limit the current promiscuous nature of the Internet, preventing a number of attack vectors which are currently exploited by web malware. We build on the joint work by providing a comprehensive description of current web attacks, and discussing extensions to the core approach.

3.1 Background

The current web environment consists of millions of servers distributed throughout the globe serving content to hundreds of millions of users worldwide. For most users, a web browser is used to fetch various pieces of content and combine them together into a web page as viewed by the user. As an example, Figure 3.1 illustrates a simple base web page. It is comprised of a number of different objects, including a graphic, text, a style declaration (which affects how the page is displayed), and JavaScript (not visible). The source is shown in Figure 3.2.

In loading a web page, the browser will first fetch a base HTML (HyperText Markup Language) page from a web server. The base page can then refer to additional files that need to be loaded in order for the page to be fully rendered to the user by the browser. The web browser will fetch all extra objects referenced by the base page as part of the process of loading the page. In the example code of Figure 3.2, this includes fetching an image (line 11), a JavaScript source file

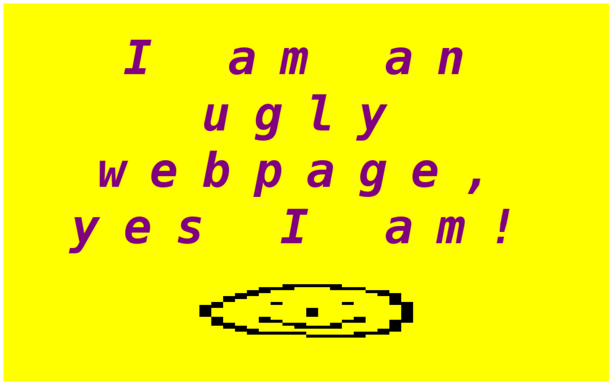

I am an ugly webpage, yes I am!

Figure 3.1. A sample web page including a graphic.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"                1
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">          2
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">               3
<head>                                                                  4
  <title>Yes?</title>                                                   5
  <link rel="stylesheet" href='style.css' type="text/css" />            6
  <script type="text/javascript" src='code.js'></script>                7
</head>                                                                 8
<body>                                                                  9
  <p>I am an <span class="blink">ugly webpage</span>, yes I am!         10
  <img alt="" src='face.png' width="400" height="100" /></p>            11
</body>                                                                 12
</html>                                                                 13
```

Figure 3.2. Source code for a sample web page including a graphic.

(line 7), and a style sheet file (line 6). In the example, none of the links include an explicit domain name, hence the objects they point to are fetched from the same server as the base page. Another possibility, however, is that each link refers to a complete URL, allowing objects referenced by the base page to exist on servers associated with different origins. For each object loaded by a web browser, the origin of that object is defined as the domain name, port number, and protocol through which the element was fetched. JavaScript is currently the most popular scripting language [100].

3.1.1 Same Origin Policy

The same origin policy we focus on in this thesis is enforced on all JavaScript code which is run within a web browser (we discuss alternate same origin policies in Section 3.7.1). It limits the ability of JavaScript to read from and modify objects retrieved from a different origin [79]. As each object is loaded, the origin of the object is stored by the web-browser as a tag in the meta-data associated with that object. We define the same origin policy as a restriction on the ability for JavaScript tagged with one origin to read from or modify objects tagged with a different origin. In other words, JavaScript tagged with origin A is not allowed to read data tagged with a different origin B (but can still display this data to the user). It also cannot modify data tagged with origin B (i.e., what is displayed to the user is the same as what was received from the server B). While JavaScript cannot modify data received by server B (because the origin is not the same), it can obscure the data as it is displayed to the user (see Section 3.1.2). Table 3.1 shows where the same origin policy applies. For each content type, the ability for JavaScript to fetch, read, modify, and execute the content is indicated. For those permissions denoted SO, JavaScript can only perform the operation if the content is tagged as having the same origin as the JavaScript attempting to perform the operation.

Content Type	**Fetch**	**Read**	**Modify**	**Execute**
Images	Yes	SO	SO	No
HTML	Yes	SO	SO	No
Raw Data	Yes	SO	SO	No
JavaScript	Yes	Yes[1]	Yes	Yes
Styles	Yes	Yes[1]	Yes	No
Audio/Video (Plugins)	Yes	Plugin Dependent		No
Audio/Video (HTML5)	Yes	SO	SO	No

Table 3.1. Current JavaScript access to content loaded by the web browser (e.g., JavaScript may always fetch images, but can only read or modify the contents of an image if is tagged with the same origin (SO) as the JavaScript attempting the operation).

[1]The contents of a style and JavaScript file can be determined by examining the effect they have on the document after being interpreted by the browser. Style elements are embedded into the document, appearing as part of the DOM. Loaded JavaScript functions can likewise be examined by other JavaScript code through the `toString()` method (unless the function has been redefined to hide its previous functionality).

In Table 3.1, the ability to modify styles and JavaScript is not limited by the same origin policy. Instead of being self-contained objects which loaded, tagged, and (potentially) displayed to the user, these two types of objects influence the look of the whole page. A style file does this directly by dictating, for each HTML element on the base page, how it should be displayed (e.g., margins, text colour, borders, etc.). JavaScript, through access to the base HTML page, can also modify the look and feel of the page (and hence overwrite any style dictated by the style file).

In our example of Figure 3.2, one of the objects loaded onto the page was JavaScript code. In general, JavaScript code can be loaded in one of two ways, either through including it directly on the base HTML page, or through loading it as an external script file. While it is obvious that code embedded on the base HTML page is tagged with the same origin as the page (as part of that base page), what is less obvious is the tagged origin of JavaScript code loaded through an external script. While in general, the tagged origin indicates the domain name, port number, and protocol through which an element was fetched, an exception is made for JavaScript. All JavaScript on a page, regardless of where it was originally loaded from, is tagged with the origin of the base HTML page. In tagging all JavaScript with the same origin as the base HTML page, all JavaScript has permission to read and modify to the base page, but not to other elements tagged with a different origin. Because all JavaScript is tagged with the same origin, any JavaScript loaded can redefine any function. This is the case even if the function was previously defined by a JavaScript source file which came from different origin.

The HTML `iframe` tag is a special element which can be used to embed one complete web page within another. It can be thought of as allowing a new base HTML page within another page. All elements (including JavaScript and styles) referenced inside the `iframe` affect the `iframe` instead of the base HTML page which contains the `iframe`. If the `iframe` element is tagged with a different origin than the base HTML page, the same origin policy applies, preventing the base HTML page from reading content, modifying content, sending messages, and reading replies from the `iframe`. Similarly, the content in the `iframe` cannot read, modify, or send messages to the base HTML page. The exception to this rule is if the base HTML page and HTML page inside the `iframe` came from different sub-domains within the same base domain (e.g., `ca.example.org` and `rp.example.org`). In this case, content from the sub-domains is allowed to communicate if and only if they both indicate their explicit desire updating the tagged origin through setting the JavaScript variable `document.domain` to the same common prefix (i.e., `example.org`).

Document Object Model

The *Document Object Model* (DOM) is the representation of the base HTML page which is made available to JavaScript [84]. It allows JavaScript to read and modify the base HTML content, as well as content referred to by the base HTML page (subject to the constraints of the same origin policy). It also allows JavaScript to associate event handlers with events that are triggered by the user interacting with the HTML elements (e.g., `onclick` or `onmouseover`). We choose not to tie the JavaScript same origin policy to *Document Object Model* (DOM) access in this thesis because it is possible to load content from a different origin without inserting it into the DOM, as illustrated in Figure 3.3. The JavaScript same origin policy restricts access to objects loaded by the web browser, regardless of whether they have been inserted into the DOM. In implementing SOMA, we extend policy dictating the fetching of content beyond JavaScript, going beyond the DOM interface.

```
var pic = new Image();
/* Assigning a value to pic.src causes the browser
   to fetch the image. */
pic.src = "http://example.com/image.png";
```

Figure 3.3. Load an image without inserting it into the DOM.

Application of the Same Origin Policy

The same origin policy restricts JavaScript's ability to read and modify objects that have been fetched by the browser. In Figure 3.4, JavaScript on base HTML page A can read and modify object B, since it comes from the same server (server 1). JavaScript on base HTML page A cannot read or modify C, since it came from server 2. The same origin policy also restricts the ability to read and modify objects associated with other windows (or tabs) which may be open concurrently by the browser. In Figure 3.4, JavaScript on base HTML page A should not be able to read or modify base HTML page D, or the objects E and F.

Asynchronous JavaScript calls (AJAX) result in the contents of an object being returned directly to JavaScript (instead of being treated as a distinct object by the browser). Because JavaScript can read and modify content returned as the result of an AJAX request, these requests are limited by the browser to being performed only to the origin of the base HTML document.

As discussed in 3.1.1, all JavaScript loaded on a page inherits the origin of the base page. All JavaScript therefore has full access to the base page

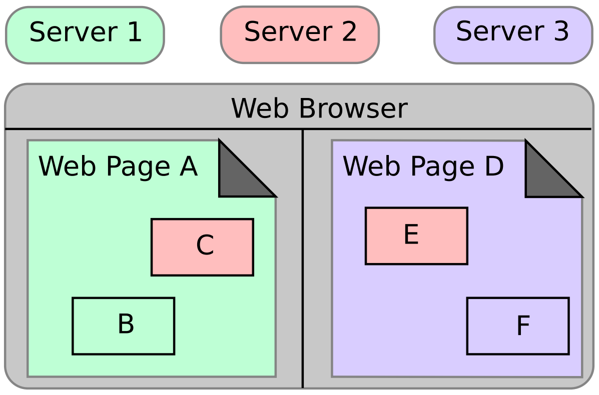


Figure 3.4. A pictorial representation of two different web pages being displayed in a browser concurrently. Page A is from server 1 and includes objects B (also from server 1) and C (from server 2). Page D is from server 3, and includes objects E (from server 2) and F (from server 3).

and any objects embedded on it that came from the same origin as the base page. Mashups involve combining content (including JavaScript) from many different sources on a single web page [74]. With the introduction of mashups, the desire to restrict the access of JavaScript to content (and other JavaScript) coming from the same origin has become an issue. We choose to focus on the currently exploited attack vector of communication between the browser and external servers in this thesis.

Benefits of the Current Same Origin Policy

By limiting the ability to read and modify content tagged with a different origin, many web attacks are prevented. The same origin policy prevents an attacker from performing the following attack (which we term a fetch-parse-fetch attack) in JavaScript: 1) Load an HTML page (or other object) from a different origin. 2) Parse the contents. 3) Craft a subsequent request to the other origin based on the parse result. This restriction is important, as it blocks an attacker from implementing in JavaScript any multi-step attack which relies on the result of a previous request to any web server (other than that with the same origin as the JavaScript) in generating the next request to that server. The results of a request can not be read (and hence parsed) in JavaScript if they came from a domain other than the origin of the base HTML page; therefore the information required to make a subsequent request based on the first results is not made available to the attacker's JavaScript.

The restrictions imposed by the same origin policy are critical to the security of JavaScript and the success of the web. Being able to run code on someone else's computer is an inherently dangerous operation. Many have recognized the harm that could come to the host from running arbitrary JavaScript code and hence have implemented strong sandbox mechanisms in an effort to limit the damage that can be caused by potentially malicious JavaScript code. The other target that can be attacked by JavaScript code running at a host, however, is other servers hosting web sites. Given that all requests to remote sites made as a result of JavaScript running in a browser can not be easily tied to the source of the JavaScript by the server receiving the request, JavaScript without the same origin policy would allow the formation of very powerful distributed botnets. This would allow attackers to attack other web sites indirectly [99]. The design of the same origin policy attempted to keep the JavaScript code associated with a web application from interfering with other sites. In doing this, they prevented attackers from offloading fetch-parse-fetch attacks against other servers on unsuspecting clients. While not all attacks were prevented by the same origin policy, there is still a substantial benefit having it in effect.

3.1.2 Limitations of the Same Origin Policy

The design of the same origin policy attempts to limit damage JavaScript can do to remote servers when running in the browser. It does not stop all attacks [99]. Remaining attacks rely on the fact that many damaging activities can be performed without being able to read the response (i.e., in attacking a web server application, it is sufficient to simply send a malicious request). We now discuss some of the web attacks which are currently prevalent and why the same origin policy does not protect against them. For many of these attacks, there exist many different and inconsistent definitions [165, 184, 38].

Cross-Site Scripting

Typically, a cross-site scripting attack is carried out as illustrated in Figure 3.5. The attacker will send the malicious content to the target server, which will replay the content to a user as part of a web page it sends. A cross-site scripting vulnerability is the result of improper sanitization of input received from users of the site by a server before the server uses it as output [33, 165, 184, 38]. A single HTML file can contain a mix of JavaScript code, HTML elements, and text; with the transition between each within the file being determined by the parser. If an attacker can modify the parse tree generated from the HTML file in ways not intended by the web site developer (e.g., turning what was supposed to be text into code, or inserting HTML elements not intended), then a cross-site scripting vulnerability exists [103].

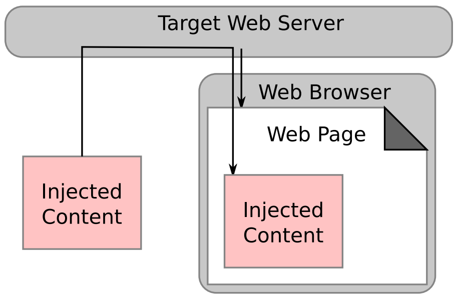


Figure 3.5. A cross-site scripting web attack.

The malicious input which exploits a cross-site scripting vulnerability can either be uploaded and stored on the server (e.g., a blog post stored on a server and shown to everyone viewing the blog) and continue to be sent to users long after the attack has taken place, or it can be passed as a parameter (e.g., a search request) to a dynamic page generated by the web server and returned to a user. Both of these methods of exploit result in additional content provided by the attacker being embedded in the web page served.

A cross-site scripting attack is not prevented by the same origin policy because there is no requirement for JavaScript to read or modify data tagged with a different origin. The malicious content sent to the server by the attacker is simply reflected as part of the resulting web page that the server sends to the client.

Cross-Site Request Forgery

A cross-site request forgery attack involves a malicious web server sending to a client browser a page that has embedded content designed to create requests to the target web server. The attack is illustrated in Figure 3.6. In general, the attack involves interaction with other sites that the user may use at the same time as they are viewing the malicious web page [16, 165, 184, 38].

The server serving the malicious web page need not be intentionally malicious. It is sufficient to compromise an otherwise benign server through attacks such as cross-site scripting (discussed in Section 3.1.2). While it is common to see the malicious content take the form of JavaScript, it does not have to. An image tag on the attacker provided page referencing the target URL is sufficient to cause the browser to perform the request (even if the image tag URL does not actually point to an image). Any object pointed to by the attacker provided page will cause the browser to perform the request. More complex versions of the attack will craft the target URL in JavaScript, allowing the web site to perform POST as well as GET requests to the target server.

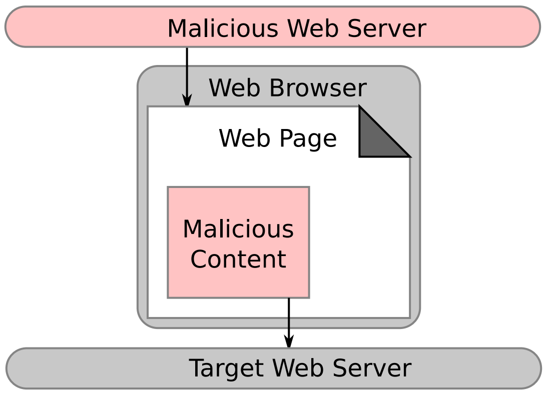


Figure 3.6. A cross-site request forgery attack.

Browsers, by design, submit cookies from a particular web server when sending another request to that same server. One common use of cookies is for storing session information, identifying a user logged onto a web site. Because all related cookies are sent with any web request to the target server, a request sent as a result of the malicious web page (even if that web page is hosted on a different origin) will also contain the appropriate cookies identifying a legitimate user. This feature makes the cross-site request forgery attack more dangerous to users.

The current JavaScript same origin policy does not prevent against cross-site request forgery for two reasons: 1) The request to the target server does not have to be performed by JavaScript, and 2) If JavaScript does perform the request, it does not need to be able to process the response (as simply making the request is enough to cause the damage). As discussed in Section 3.1.1, the same origin policy does not limit the ability to make requests.

Clickjacking

The clickjacking attack (also referred to as UI redressing and not to be confused with click fraud [186]) attempts to convince the user to perform some action against a target site while making it appear as though the action is being performed on some other unrelated site. Sections of the web page delivered to the browser by the target server are obscured by content from the malicious site, leaving visible only those target page elements the attacker wishes the user to see or interact with [10]. In hiding content from the target web page, the malicious site may be able to persuade the user to perform an otherwise unwanted task.

Since elements of the target page are not modified or read by JavaScript (they are instead covered up by the attacker), the JavaScript same origin policy

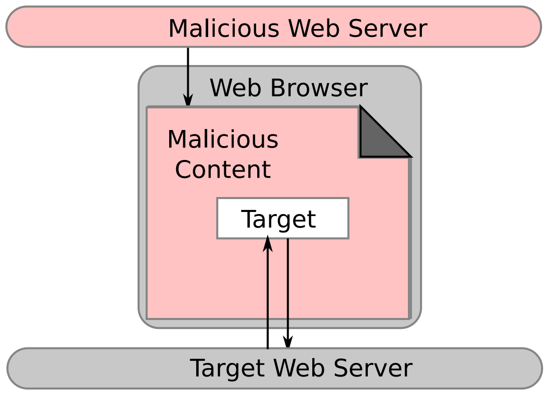


Figure 3.7. A clickjacking attack.

does not protect against this attack. The clickjacking attack is similar to the cross-site request forgery attack discussed in Section 3.1.2 in that it results in undesired web requests to the target server.

Information Stealing

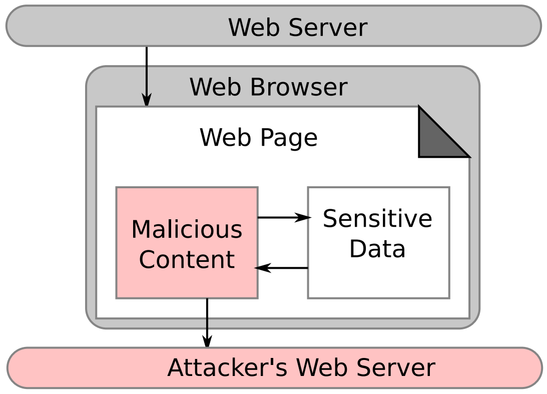


Figure 3.8. An information stealing attack.

The goal of an information stealing attack, as illustrated in Figure 3.8, is to send sensitive information to an attacker-controlled server [89]. Malicious content is injected onto a web page (e.g., through a cross-site scripting attack) which contains (or collects) sensitive information the attacker is interested in (e.g., authentication cookies, passwords, user names, account details, etc.). This malicious content can be inserted as the result of a cross-site scripting attack (Section 3.1.2) or other attack. The malicious JavaScript examines elements in the DOM of the page (e.g., form fields, and cookies) to obtain sensitive data (because both the malicious JavaScript and sensitive data originate from the same origin, the malicious JavaScript has read access). The JavaScript then sends this sensitive information to an attacker controlled server (similar to Section 3.1.2).

Bandwidth Stealing

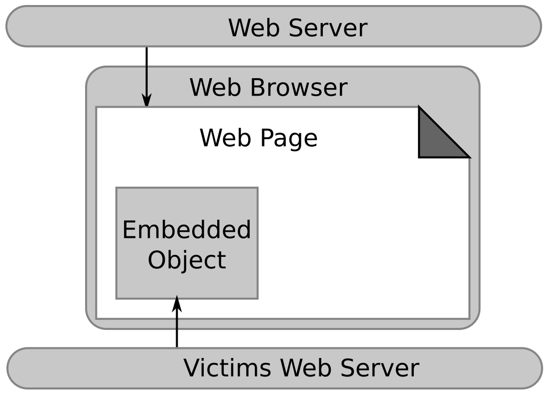


Figure 3.9. A bandwidth stealing attack.

While not strictly a classic security threat, the bandwidth stealing attack illustrated in Figure 3.9 uses the resource a victim has paid for or otherwise obtained without properly compensating them [91]. Usually, this attack takes the form of showing images or multimedia from a victim website without also showing advertisements designed to fund the continual operation of the site. In the extreme, a bandwidth stealing attack leads to a distributed denial of service attack on the victim server, due to a potentially high volume of traffic.

3.1.3 The Same Origin Policy as a Guardian MAC

The JavaScript same origin policy does not rely on the user for enforcement and is applied uniformly to all developers who create web applications. It is thus a guardian mandatory access control. Even though the same origin policy (discussed in Section 3.1.2) does not protect against all web-based attacks, there are still significant security advantages (as discussed in Section 3.1.1) which make the same origin policy worthwhile.

3.2 The Protection Mechanism

Published in 2008, the Same Origin Mutual Approval (SOMA) policy [126] is designed to strengthen the same origin policy. It does this by addressing the lack of access control surrounding fetching external content. While the same origin policy focuses exclusively on restricting the ability for JavaScript to read and modify any object tagged with a different origin, the ability to fetch content from servers is unrestricted, leading to those attacks discussed in Section 3.1.2. While the ability to read and modify content programmatically is specific to JavaScript, content can be fetched as a result of a number of different operations. SOMA restricts fetches regardless of how they are initiated, be it as the result of a JavaScript operation, HTML tag, or style tag.

Being a guardian MAC, SOMA is designed to be enabled by site administrators and enforced by the browsers, beyond the direct control of web developers. At the browser end, the policy is enforced transparently by any web browser understanding SOMA.

3.2.1 Threat Model

Our threat model for SOMA is the same as was presented in the 2008 work [126]. We assume that site administrators have the ability to create and control the content associated with top-level URLs (static files or scripts) and that web browsers will follow the policy specified at these locations correctly. In contrast, we do assume that the attacker controls arbitrary web servers and can inject content on legitimate servers through attacks such as cross-site scripting. We assume attackers are not able to alter policy files or software on legitimate servers. A goal of SOMA is to restrict communication with a malicious web server when a legitimate web site is accessed, even if the content on that site or its partners has been compromised. A related goal is to restrict communication with a legitimate web server when the user is browsing pages from an attacker controlled server.

By these assumptions, SOMA is not designed to address situations where an attacker compromises a web server to change policy files, compromises a web browser to circumvent policy checks, or performs man-in-the-middle attacks to intercept and modify communications; nor the problem of users visiting malicious web sites directly, say as part of a phishing attack. While these are all important types of attacks, by focusing on the problem of unauthorized communication between web servers and the browser, SOMA creates a simple, practical solution that works toward addressing all threats discussed in Section 3.1.2. Mechanisms to address other threats (e.g., Blueprint [103] and the `Origin` header [16]) largely complement rather than overlap with the protections of SOMA.

3.2.2 Manifest Files

SOMA is composed of two parts working together to provide the mutual approval aspects of SOMA. The first part we discuss is the SOMA manifest. This is a file fetched from the same server A as the base HTML page. It lists all servers which A authorizes as sites from which objects may be fetched during the process of building and rendering any HTML page served by A. The idea of restricting communication to a few listed external servers is borrowed from Tahoma [40]. Any server hosting objects that are referenced (either directly or indirectly) and loaded during the course of viewing the page must be listed in site A's SOMA manifest file. Any server origin not listed will not be contacted during loading of the page – any requests for objects from the unlisted origin will return an error. By convention, we assume the origin of the base HTML page is implicitly included in the SOMA manifest. We use the notation $A \text{-ⓒ-} B$ to indicate that the base page origin A authorizes that B be contacted during the course of viewing the base page which came from A. If A does not authorize communication with B during the course of viewing a web page (i.e., B is not listed on a SOMA manifest coming from A), then we denote this $A \not\text{-ⓒ-} B$.

As a standard, the paper proposed that the SOMA manifest always be located on any given server at `/soma-manifest` [126]. For a base HTML page fetched from `http://www.example.com`, the complete URL to the associated SOMA manifest would be `http://www.example.com/soma-manifest`. The actual manifest would contain a header line identifying the file as containing a SOMA manifest, followed by zero or more lines dictating other origins (recall, from Section 3.1, that an origin is defined as protocol, DNS name, and port) which can be contacted while rendering or viewing the page. See Figure 3.10.

```
SOMA manifest
http://www.example.net
http://www.example.org
```

Figure 3.10. A sample SOMA manifest file for `www.example.com`.

If the origin of the base page chooses not to implement SOMA manifests, the request for `/soma-manifest` will result in an error (e.g. 404 - file not found) being returned by the server, or a file which is not a SOMA manifest (i.e., does not contain a valid SOMA manifest header) being returned. For backward compatibility, no request for content will be blocked if the SOMA manifest does not exist. Table 3.2 indicates the possible scenarios.

Manifest	**Result**	**Interpretation**
Does not exist	$A \mathrel{-\!\subset\!\!\circ\!-} B$	All origins can be contacted
Exists, is invalid	$A \mathrel{-\!\subset\!\!\circ\!-} B$	All origins can be contacted
Exists, is valid, and does not list B	$A \mathrel{-\!\not\subset\!\!\circ\!-} B$	B cannot be contacted
Exists, is valid, and lists B	$A \mathrel{-\!\subset\!\!\circ\!-} B$	B can be contacted

Table 3.2. SOMA manifest scenarios while browsing a page with base origin A and included content from origin B.

3.2.3 Approval Requests

The second part of SOMA involves querying all domains hosting objects referenced by the page which is being viewed in the browser. SOMA queries each domain other than the origin, for permission to fetch the content referenced by the page. The approach is similar to the Adobe Flash `crossdomain.xml` [3] but differs in that SOMA returns a single `YES` or `NO` response for any given query instead of a list of origins which are allowed to include the content. Returning a single `YES`/`NO` response prevents the easy disclosure to an attacker of all sites authorized by a particular origin. We choose to return a list as opposed to a `YES`/`NO` response for SOMA manifests because similar information can easily be gleaned in most cases by parsing the base HTML page for a list of references. We use the notation $B \mathrel{-\!\circ\!\!\supset\!-} A$ to indicate that B allows its objects to be fetched for inclusion on base pages originating from A. If B does not wish to have its content fetched while a browser is rendering a page from A, we denote this as $B \mathrel{-\!\circ\!\!\not\supset\!-} A$.

```php
<?php
print "SOMA approval\n";
$policy = array(
    'www.example.com' => 'YES',
    'www.example.org' => 'YES' );
if( isset($policy[$_GET['d']]) ) {
    print $policy[$_GET['d']];
} else {
    print 'NO';
}
```

Figure 3.11. Source code for a simple SOMA approval script.

To indicate that A is authorized by B to include objects from B on its web pages, B needs to provide a script accessible which will answer `YES` when queried with the domain A. As a standard, we propose that the actual query to B be for the `/soma-approval` script and the appended key-value pair be the key `d` and the value which is the domain for which approval is being sought. For a request for approval from `www.example.net` to have its content embedded in pages coming from `www.example.com`, the complete URL of the SOMA approval request would be `http://www.example.net/soma-approval?d=www.example.com`. A simple SOMA approval script which would respond to such requests is illustrated in Figure 3.11.

If the administrator of the server hosting the content which is to be embedded in pages chooses not to implement SOMA, the SOMA approvals script will not exist and the server will return a 404 (file not found) or some other error for any approval request. If the server administrator chooses to use the `/soma-approval` script for an unrelated purpose, the header line will not be `SOMA approval` and the browser should treat this the same as an error response. In both cases, we assume that the administrator of the server from which the content is being fetched for inclusion allows the fetches to take place (so the default is backwards compatible). Table 3.3 lists the possible scenarios when attempting to fetch a SOMA approval.

Approval Script	**Result**	**Interpretation**
Does not exist	$B \mathrel{-\!\odot\!-} A$	Objects can be embedded on any page
Exists, is invalid	$B \mathrel{-\!\odot\!-} A$	Objects can be embedded on any page
Exists, responds with `NO`	$B \not\mathrel{-\!\odot\!-} A$	A's pages cannot embed B's objects
Exists, responds with `YES`	$B \mathrel{-\!\odot\!-} A$	A's pages can embed B's objects

Table 3.3. SOMA approval scenarios while browsing a page with base origin A and included content from origin B.

3.2.4 The Combination of Manifests and Approvals

One of the design features of SOMA is that it enforces mutual approval for content inclusion on a web page. If either the administrator for the base page origin or the administrator for the included content origin state that they disallow objects being fetched, then the SOMA-supporting browser will honour this and refuse to fetch the objects. Only if both parties agree is content fetched and included into a web document. Table 3.4 indicates the different possibilities and when the content is actually fetched.

Manifest	**Approval**	**Result**
$A \mathrel{-\!\not\textcircled{c}\!-} B$	$B \mathrel{-\!\not\textcircled{ɔ}\!-} A$	Objects are *not* fetched from B
$A \mathrel{-\!\not\textcircled{c}\!-} B$	$B \mathrel{-\!\textcircled{ɔ}\!-} A$	Objects are *not* fetched from B
$A \mathrel{-\!\textcircled{c}\!-} B$	$B \mathrel{-\!\not\textcircled{ɔ}\!-} A$	Objects are *not* fetched from B
$A \mathrel{-\!\textcircled{c}\!-} B$	$B \mathrel{-\!\textcircled{ɔ}\!-} A$	Objects are fetched from B

Table 3.4. SOMA manifest and approval combinations while browsing a page with base origin A and included content from origin B.

The pseudo-code process a browser actually follows in fetching a complete page when SOMA is being enforced is shown in Figure 3.12. The pseudo-code illustrates the parallelizable nature of SOMA-related requests. When building a web page in the browser with SOMA enabled, the base HTML page and SOMA manifest are fetched in parallel. Then, for each element from a different origin referenced by the base HTML page, a SOMA approval request is sent. Only when an affirmative answer is received to the SOMA approval request is the actual request for the object sent to the remote server.

3.3 Benefits

SOMA provides some protection against the current web threats as discussed in Section 3.6.4. The approach does not rely on end-users to know anything about the policy, or that they be involved in enforcement of the policy. This is accomplished through building the enforcement mechanism directly into the browser, similar to how the JavaScript same-origin policy works. We do not recommend a potential extension to the core SOMA approach that would allow end-users to modify either the SOMA manifest or approval responses.

SOMA policies are set up by system administrators who maintain the web servers as opposed to developers who write web applications. In separating SOMA policy from the web application development, we remove control from the web developers and place it in the hands of the system administrator. The underlying idea here is that the system administrator likely has more of a vested interest in the security of their web site than the developer of the web application, who may be disconnected from the environments it may be used in. We now discuss several other advantages.

```
function origin( URL ) {
  return URL.proto + URL.domain + ':' + URL.port + '/';
}

function buildPage( URL ) {
  object approvals = array();
  object manifest = async_fetch( origin(URL) + 'soma-manifest' );
  object base = sync_fetch( URL );

  while( size_of(base.unfetched_objects) != 0 ) {
    /* Process each object referenced by the base page */
    curobj = unqueue(base.unfetched_objects);

    /* Does the manifest allow us to talk to the server? */
    wait_for( manifest );
    if( manifest.allows(origin(curobj.URL)) == 'NO' ) {
      curobj.data = NULL;
      next;
    }

    if( approvals[origin(curobj.URL)].inProcess() ) {
      /* Waiting for approval response, check back later. */
      queue( base.unfetched_objects, curobj );
    } elif( approvals[curobj.URL.domain].allowed() == 'YES' ) {
      /* Server allows its content to be embedded. */
      curobj.data = async_fetch( curobj.URL );
    } elif( approvals[curobj.URL.domain].allowed() == 'NO' ) {
      /* Server does not allow its content to be embedded. */
      curobj.data = NULL;
    } else {
      /* Query server - Can its content can be embedded? */
      approvals[origin(curobj.URL)] =
        async_fetch( origin(curobj.URL) + 'soma-approval?d=' +
                     base.domain );
      /* Requeue request until we get our approval answer */
      queue( base.unfetched_objects, curobj );
    }
  }
}
```

Figure 3.12. Pseudo-code for the browser SOMA enforcement process.

3.3.1 Backwards Compatibility

The lack of a SOMA manifest file defaults to a blanket accept (and likewise the lack of a SOMA approval script defaults to allowing the fetch). With lack of a SOMA policy being interpreted as an allow, any site choosing not to implement SOMA will continue to work as it does currently. Because SOMA does not replace the JavaScript same origin policy but instead expands it, security is not reduced compared to the current web if one chooses not to use SOMA. For browsers not implementing SOMA, SOMA manifest or approval files will simply not be accessed on the server. For these browsers, again the web will continue to work as it currently does. Only when both the server and browser support SOMA will the protections start to be enforced. This approach allows incremental deployment with incremental benefit.

In dictating that the first line of a SOMA manifest or approval response be a header containing “SOMA manifest” or “SOMA approval”, we can differentiate between responses from servers not understanding SOMA but responding with generic response pages, and those servers understanding and responding to SOMA requests.

Complete SOMA protection is composed of three elements: a browser enforcing SOMA, a SOMA manifest file at the base page origin, and a SOMA approval file for each domain contacted during the course of rendering a web page. If the browser does support SOMA but a manifest does not exist at the base site, the SOMA enabled browser will still query the SOMA approval script associated with each object included on the web page and adhere to the restrictions returned by the script. Likewise, a base origin distributing a SOMA manifest will have its protections enforced regardless of whether the included servers provide a SOMA approvals file. While it is possible to deploy SOMA manifests without SOMA approvals (or vice-versa), deploying both provides maximum benefit.

3.4 Limitations

We now discuss some limitations of the SOMA approach.

3.4.1 Third Party Advertisements

Ad syndication involves allowing an advertiser to sell advertising space to other advertising companies. Under SOMA, if site A decides to sell advertising space on its site to advertiser B, then site A will typically list B on its SOMA manifest.

Likewise, B will indicate that A is allowed to embed its content in the SOMA approvals script. If advertiser B turns around and wants to sell its space on site A to C, then C will not be listed in the SOMA manifest for A and hence SOMA will not allow its content to load. In such a case, A must either add C to the SOMA manifest, allowing content from C to appear on the page coming from A, or B must somehow proxy the data so that all ads appearing on the site still come from B (even though B has sold the advertising space to C). Unless B sets up a proxy so that it appears all content being included on the page hosted by A comes directly from B, the presence of a SOMA manifest on A will result in B being restricted in their ability to perform ad syndication.

We do not see the limitation on ad syndication as being a negative limitation of SOMA. Indeed, the practice of ad syndication has contributed significantly to the rise in ad delivered malware. In fact, multiple levels of ad syndication are used in 75% of all ad delivered malware [135]. The practice of using multiple levels of ad syndication is made difficult through the introduction of SOMA.

3.4.2 External Communication

SOMA is designed to improve the same origin policy by imposing further constraints upon external inclusions and thus external communications. It does not prevent attacks that do not require external communications such as code and content injection. SOMA can restrict outside communication frequently seen in current attack code [135].

SOMA does not stop attacks to or from mutually approved communication partners. In order to avoid these attacks, it would be necessary to impose finer grain control or additional separation between components. This sort of protection can be provided by the mashup solutions described in Section 3.7.4, albeit at the cost of extensive and often complex web site modifications.

SOMA cannot stop attacks on the origin where the entire attack code is injected, if no outside communication is needed for the attack. This includes attacks such as web page defacement, some forms of cross-site scripting, or sandbox-breaking attacks intended for the user's machine. Some complex attacks might be stopped by size restrictions on uploaded content. More subtle attacks might need to be caught by heuristics used to detect cross-site scripting. Some of these solutions are described in Section 3.7.

SOMA cannot stop attacks from malicious servers not including content from remote domains. These would include phishing attacks where the legitimate server is not involved.

3.4.3 Self-Contained Malicious Servers

If a malicious server does not serve web pages that rely on non-malicious servers in order to perform an attack (e.g., the malicious server simply hosts a phishing attack where all images and other data originate from the attacker and the data obtained from phishing is sent back to the attacker), then SOMA will not help. SOMA is designed to limit communication in the scenario where at least one of the servers involved is non-malicious.

3.5 Extensions to the Core Approach

While the core SOMA approach goes a long way to improving the security of the web over the same origin policy, there are still improvements that can be made over the core approach.

3.5.1 Third Party Provided Manifests and Approvals

The core idea relies on every server who wishes to take advantage of the protections offered by SOMA to host either a manifest file, approvals script, or both on their servers. There may be, however, sites that would greatly benefit from the deployment of SOMA even if the site administrator is slow to adopt the technology. In this case, it becomes beneficial for an external third party to be able to host manifest files and approval scripts on behalf of the site which is to be protected by SOMA.

To support the use of third party SOMA manifest and approval servers while still not requiring the end-user to administer the scheme, a specific site could be chosen to host these files (e.g. `http://soma.net`). This server would be tasked with responding with the correct SOMA manifest file given the base origin (including protocol, DNS name, and port) and respond to any SOMA approval request (again, given the base domain and included content origin). One way of accomplishing this is with a directory structure that includes the protocol, host name, and port as elements. As an example, a request for a SOMA manifest for `http://www.example.com` from the third party would be translated by the browser into a request for `http://soma.net/http/www.example.com/80/soma-manifest`. The same approach can be used for the SOMA approval requests, where a request for a SOMA approval from `http://www.example.net` for content inclusion on a base page from `www.example.com` would be translated by the browser to a request for `http://soma.net/http/www.example.net/80/soma-approval?d=www.example.com`. Like with a request for a SOMA approval or manifest file directly from the site, an error response from the third party server would initiate the fallback of allowing all fetches.

If sites were to continue to implement SOMA, the situation is likely to occur where a SOMA manifest (or approval) is hosted on both the directly involved site as well as the third party. When this situation occurs, we stipulate that the SOMA manifest (or approval) on the real site should be followed as opposed to that hosted on the third party site.

3.5.2 Protocol and Port in the Approval Script

The core SOMA approach does not send the protocol or port to the SOMA approval script when asking for approval to include content on a page from a different origin. While at this point we see no reason why the port or protocol would be required by the approval script, the design of the interface to the approval script makes expansion to deal with these attributes trivial. Additional options (mainly, t=<protocol> and p=<port>) can be appended to the SOMA approval request (e.g., for an approval request from `http://www.example.net` approving `http://www.example.com`, the request would be `http://www.example.net/soma-approval?d=www.example.com&t=http&p=80`). As an optimization, if the protocol is using the standard port, the port number can be omitted in the SOMA approval request.

3.5.3 SOMA Implemented in HTTP Headers

In our proposal, a request for a SOMA manifest is done in parallel with a request for the main page. A request for a SOMA approval needs to be done *before* the request for the subsequent content from the server. The reasons for this are two-fold.

1. By separating the manifest (and approval) from the object being returned (i.e., either the base page or included content), a different cache policy can be set for each by the web server administrator. As an example, the base page may change frequently while the SOMA manifest does not change. By separating the two, the cache lifetime of the SOMA manifest or approval can be set such that it does not need to be retransmitted alongside each request for a new object from the same web server.

2. By separating the SOMA manifest and approval requests from the request for actual content, the web developer does not need to ensure that their

web application always sends a manifest or approval response alongside the desired object. One of the goals of SOMA is to place control of manifests in the hands of the server administrator, not the web developer. While the request for the object could be separated from the SOMA manifest/approval request by the web server, the implementation of such approach is more involved than simply making a `/soma-manifest` file available.

Because one of the benefits of the SOMA approval is that it protects against cross-site request forgeries (see Section 3.6.4), one must ensure that the actual request is not received by the web application running on the server until *after* the SOMA approval has been granted. To ensure that the SOMA approval is processed before the request for the content, we separate the two.

While the implementation of SOMA as HTTP request/response headers can potentially lead to performance improvements due to a reduction in the number of round-trips required to load a web page, the implementation of such an approach should be designed with care. Should one decide to implement SOMA as HTTP headers alongside the traditional object request, we suggest that the processing of SOMA related headers be performed by the core web server (e.g., Apache), not the web application running alongside the web server. In doing this, web developers remain shielded from having to properly process and respond to SOMA related requests.

3.6 A Prototype Implementation

In order to test SOMA, we created an add-on for Mozilla Firefox 2.0, licensed under the GNU GPL version 2 or later.[1] It can be installed in an unmodified installation of Mozilla Firefox the same way as any other add-on: the user clicks an installation link and is prompted to confirm the install. If they click the install button, the add-on is installed and begins to function after a browser restart.

The SOMA add-on provides a component that does the necessary verification of the `soma-manifest` and `soma-approval` files before content is loaded.

Since it was not possible to insert test policy files onto sites over which we had no control, we used a proxy server to simulate the presence of manifest and approval files on popular sites. We now discuss the deployment costs, any compatibility problems encountered, performance, and protection against current web attacks.

[1] See `http://ccsl.carleton.ca/software/soma`

3.6.1 Deployment Costs

The browser, the origin sites, and content inclusion provider sites all bear the costs in deploying SOMA. Note that unlike solutions that rely heavily upon user knowledge (e.g., the NoScript add-on for Mozilla Firefox [106]), SOMA requires no additional effort on the part of the user browsing the web site. Instead, policies are set by server operators, who are expected to have more information about what constitutes good policy for their sites.

Deployment in the Browser

The SOMA policy is enforced by the web browser, so changes are required in the browser. The prototype SOMA implementation, as used in the paper 3, was deployed as an add-on. The prototype SOMA add-on, when prepared into the standard XPI package format used by Mozilla Firefox, is 16kB. Uncompressed, the entire add-on is 18kB. The component that does the actual SOMA mutual approval process is 12kB. The SOMA add-on prototype provides a persistent visual indication that it is loaded in the bottom-right corner of the browser window. This visual cue was used during the development of the prototype, but could be hidden on production deployments of SOMA.

Deployment on Origin Sites

Each origin server that wishes to benefit from the protections of SOMA needs to provide a `soma-manifest` file. This is a text file containing a list of content-providing sites from which the origin wishes to allow included content. As mentioned earlier, each origin is specified by a domain name, protocol and (optionally) port.

This list can be determined by looking at all pages on the site and compiling a list of content providers. This could be automated using a web crawler, or done by an admin who is willing to set policy (it is possible that sites will wish to set more restrictive policy than the site's current behaviour). In the paper [126], the main page on popular sites was examined to determine the approximate complexity of manifests required. The PageStats add-on [48] to load the home page for the global top 500 sites as reported by Alexa [4] and examined the resulting log, which contains information about each request that was made. On average, each site requested content from 5.45 domains other than the one being loaded, with a standard deviation of 5.3. The maximum number of content providers was 32 and the minimum was 0 (for sites that only load from their own domain).

Of course, a site's home page may not be representative of its entire contents. As a further test, the paper documents the traversal of large sections of a major Canadian news site (www.cbc.ca). The number of domains needed in the manifest was approximately 45; this value was close to the 33 needed for that particular site's home page.

Given the remarkable diversity of the Internet, sites probably exist that would require extremely large manifest files. Cursory exploration documented in the paper, however, gives evidence that manifests for common sites would be relatively small.

Deployment on Content Provider Sites

Content providers wishing to take advantage of SOMA need to provide either a file or script that can handle requests to soma-approval. The time needed to create this policy script depends heavily upon the needs of the site in question, and may range from a simple yes-to-all or no-to-all to more complex policies based upon client relationships. Fortunately, simple policies are likely to be desired by smaller sites (which are unlikely to have the resources to create complex policies), and complex policies are likely to be required only by larger more connected sites.

Many sites will not wish to be external content providers and their needs will be easily served by a soma-approval file that just contains NO. Such a configuration will be common on smaller sites such as personal blogs. It will also be common on high-security sites such as banks, who want to be especially careful to avoid cross-site request forgery and having their images used by phishing sites (phishing sites can still copy over images as opposed to linking to the original image).

Other sites may wish to be content providers to everyone. Sites such as Flickr and YouTube that wish to allow all users to include content will probably want to have a simple YES policy. This is accomplished by either having soma-approval always return YES, or by not hosting a soma-approval file (as the default is YES).

The sites requiring the most configuration are those who want to allow some content inclusions rather than all or none. For example, advertisers might want to provide code to sites displaying their ads. The domains that need to be approved can be determined using the list of domains already associated with each clients profile. This database could then be queried to generate the approval list. Or a company with several web applications might want to keep them on separate domains but still allow interaction between them. Again, the necessary inclusions will be known in advance and the necessary policy could be created by a system administrator or web developer.

The paper documents the overhead of soma-approval requests by using data from the top 500 Alexa sites [4], 3244 cases in which a content provider served data to an origin site are examined [126]. The time frame for these tests was April 2008. The average request size was 10459 bytes. Because many content providers are serving up large video, the standard deviation was fairly large: 118197 bytes. The median of 2528 bytes is much lower than the average. However, even this smaller median dwarfs the ≈ 20 bytes required for a `soma-approval` response. As such, we feel it safe to say that the additional network load on content providers due to SOMA is negligible compared to the data they are already providing to a given origin site.

3.6.2 Compatibility with Existing Web Pages

To test compatibility with existing web pages, the global top 45 sites as ranked by Alexa [4] were visited in the browser with and without the SOMA add-on [126]. No SOMA compatibility issues were detected. These results were expected, as SOMA was designed for compatibility and incremental deployment.

3.6.3 Performance

Drawing from the paper [126] for the performance analysis, the primary overhead in running SOMA is due to the additional latency introduced by having to request a `soma-manifest` or `soma-approval` from each domain referenced on a web page. While these responses can be cached (like other web requests), the initial load time for a page is increased by the time required to complete these requests. The manifest can be loaded in parallel with the origin page, and so we do not believe manifest load times will affect total page load times. Because `soma-approval` files must be retrieved before contacting other servers, however, overhead in requesting them will increase page load times.

Since sites do not currently implement SOMA, SOMA's overhead was estimated using observed web request times. First, the average HTTP request round-trip time for each of 40 representative web sites was determined[2] on a per-domain basis using PageStats [48]. The per-domain average was used because a proxy for the time to retrieve a `soma-approval` from a given domain. Then, to calculate page load times using SOMA, the time to request all content from each accessed domain by the `soma-approval` request was estimated for that domain. The time of the last response from any domain then serves as the final page load time.

[2]The representative sample included banks, news sites, web e-mail, e-commerce, social networking, and less popular sites.

After running our test 30 times on 40 different web pages, the paper documents the average additional network latency overhead due to SOMA as increasing page load time from 2.9 to 3.3 seconds (or 13.28%) on non-cached page loads. On page loads where `soma-approval` is cached, the overhead is negligible. This increase in latency is due to network latency and not CPU usage. If 58% of page loads are assumed to be revisits [164], the average network latency overhead of SOMA drops to 5.58%. We expect that this overhead could further drop should SOMA be implemented within the HTTP headers (as discussed in Section 3.5.3).

Given that `soma-approval` responses are extremely small (see Section 3.6.1), they should be faster to retrieve than the average round-trip time estimate used in our experiments. These values should therefore be seen as a worst-case scenario. In practise, we expect SOMA's overhead to be significantly less.

3.6.4 Protection Against Current Attacks

In order to verify that SOMA actively blocks information leakage, cross-site request forgery, cross-site scripting, and content stealing, examples of these attacks were created. In the paper 3, the SOMA add-on was specifically tested with following attacks:

1. A GET request for an image on another web site (testing both GET based cross-site request forgeries as well as content stealing).

2. A POST request to a page on another web site done through JavaScript (testing POST based cross-site request forgeries).

3. An `iframe` inclusion from another web site (testing `iframe` injection based cross-site scripting).

4. Sending either a cookie or personal information to another web site (testing information leakage).

5. A script inclusion from another web site (testing a bootstrap cross-site scripting injection).

All attacks were hosted at domain A and used domain B as the other domain involved. All attacks were successful without SOMA. With SOMA, these attacks were all prevented by either a manifest at domain A not listing B or a `soma-approval` at domain B which returned `NO` for domain A. We now discuss in more detail how SOMA works to block each type of attack.

Cross-Site Scripting Bootstrap

While the most general form of a cross-site scripting attack (as discussed in Section 3.1.2) does not result in communication with any external sites as the compromised page is viewed, a subset of cross-site scripting attacks do involve loading additional code from a remote URL. For those attacks the cross-site scripting exploit is a multi-step process.

The first step involves finding and exploiting a cross-site scripting vulnerability as discussed in Section 3.1.2. Because many cross-site scripting vulnerabilities only result in an attacker being able to upload very short snippets of JavaScript, a common approach is to use the code embedded into the page through the vulnerability as a bootstrap. The bootstrap code loads another longer script which can perform the complex operations that the attacker may need to perform to take full advantage of the cross-site scripting vulnerability. This secondary script is often located on a different external server that the attacker can easily host scripts on.

The second step of exploiting a cross-site scripting vulnerability through loading additional JavaScript from a different external site is limited by the SOMA manifest in the following way. If the site which hosts the additional JavaScript is not listed on the manifest, then the bootstrap process will fail. In order to exploit a cross-site scripting vulnerability for a site deploying SOMA, the attacker must either embed all the JavaScript attack code in the server request being exploited to perform the attack (assuming the attack code is short enough to not be truncated/rejected), or the additional JavaScript must be hosted on a site listed in the SOMA manifest file.

Unrestricted Outbound Communication

As discussed in Section 3.1.2, the ability to scrape potentially sensitive information from a web page (including form input) and send it to a server controlled by the attacker is a current threat. Similar to how SOMA defends against the cross-site scripting attack, this attack is limited through the use of a SOMA manifest. If the manifest file does not list the attacker's site as one of the authorized external sites, the attempt to send information to the attacker's server will fail (since no communication will occur between the browser and attacker controlled server unless it is on the manifest).

Recursive Script Inclusion

Similar to the cross-site scripting vulnerability listed in Section 3.6.4, a recursive script inclusion involves a legitimate script loading another script that the developer of the web application did not intend to be loaded. While the author of site A may refer to scripts on domain B, the author may not wish site B to turn around and load scripts from site C.

A common use of recursive script inclusion is ad syndication, as discussed in 3.4.1. Ad syndication has been used in the past as a vector to compromise web sites, and hence developers may want to enforce that ad syndication remain disallowed. To disallow ad syndication, the developer of a web site creates a SOMA manifest specifying the ad server - forcing all ad elements to come directly from the ad server, and not through other external sites.

Drive-By Downloads

A typical drive-by download is initiated when a victim uses their browser to browse a landing web page that is malicious. This landing page may be intentionally malicious (e.g., visiting the attacker's site directly) or it may have become malicious as a result of being compromised by an attacker. Typically, sites which have been compromised by an attacker are much more likely to receive high volumes of traffic than sites hosted by the attacker directly (e.g., the Dolphin Stadium website was compromised by an attacker during the Super Bowl in 2007, embedding a link to malicious JavaScript [185]). The malicious activities of the landing page are typically hidden in an attempt to keep the malicious actions undetected. To hide the malicious activities of the landing page, a single reference to a URL will be embedded in it, normally in the form of a `script` or `iframe` tag, leaving the rest of the landing web page as it had appeared before the compromise. Furthermore, malicious content pointed to by the URL will not typically be hosted on the same server as the landing page [135].

For those malicious landing pages that have been compromised by the attacker, SOMA provides a defence. If the site hosting the landing page also hosts a SOMA manifest, the malicious content would have to be hosted on a site listed in the manifest for the attack to succeed. If the malicious content is not hosted on a site listed in the SOMA manifest, browsers enforcing SOMA will not fetch the malicious content, causing the attack to fail.

The drive-by download attack is very similar to the cross-site scripting attack discussed in Section 3.6.4. Both refer to malicious content through a URL embedded on the site and both can be protected against through the use of a SOMA manifest (not listing the attackers server) and SOMA enabled browser.

Cross-Site Request Forgery

As discussed in Section 3.1.2, a cross-site request forgery results in requests to a victim site that the web browser user did not intend. These requests, however, are submitted as part of loading or viewing a page that came from a different origin.

SOMA prevents cross-site request forgery when an approval script is used at the victim site. Any request to the victim will be prefixed with a request to the approvals script, specifying the origin of the page which is causing the request. For a cross-site scripting attack, the malicious host would be specified in the request to the approvals page. In order to prevent the subsequent cross-site request forgery, the victim website needs only to answer `NO` to the approvals request.

Clickjacking

As discussed in Section 3.1.2, a clickjacking attack involves covering up content on a web page delivered from the target server with content originating from an attacker. If the base web page is delivered from an attacker controlled server and content from the target is embedded into it, clickjacking can be mitigated through the use of a SOMA approval script on the target web server. If the base web page is delivered from the target web server and the additional malicious content comes from an attacker controlled server, then the use of a SOMA manifest file which does not include the malicious server as an accepted origin will mitigate the clickjacking attack.

Bandwidth Stealing

As discussed in Section 3.1.2, bandwidth stealing results in content being embedded in a web page against the wishes of those responsible for the servers on which the content is hosted. SOMA protects against bandwidth stealing in a way similar to blocking cross-site request forgeries (Section 3.6.4). Any request to embed content in a site from a different origin will result in a request to the victim site for approval. To avoid having content embedded in pages coming from a different origin, the victim need only answer `NO` when queried by a SOMA enabled browser.

3.7 Related Work

Because modern browsers are capable of browsing multiple sites concurrently, the objects associated with one site in the browser must be properly segregated from objects related to another in order for the same origin policy to be properly enforced. Chen et al. [34] examined gaps in current implementations of the same origin policy related to the ability to view multiple pages concurrently in a browser. They proposed a script accenting mechanism to mark data as associated with a particular web page, using it to prevent data leakage between different web pages being viewed within the browser. Barth et al. [17] also proposed an algorithm for detecting illegitimate sharing of content within the browser. Jackson et al. [73] also focused on communication within the browser, examining how web pages can gain information about other web pages which have been viewed in the browser and how to restrict such information leakage. Ries et al. [137, 138] proposed filtering known-malicious JavaScript at the client before it is processed by the web browser. Their approach focuses on JavaScript which is known to exploit browser bugs. Unlike SOMA, these approaches focus on local rather than remote communication.

3.7.1 Same Origin Policies in Other Domains

The term “same origin policy” is one which has been overloaded, referring to different things depending on the context [194]. While we use it in this thesis to refer to restrictions placed on the ability of JavaScript to read and modify objects tagged with a different origin, there are several other definitions worth mentioning.

Cookie Same Origin Policy

Cookies do not have a same origin policy according to the formal descriptions of them [94, 95]. Regardless, some have termed the policy which dictates a browser’s handling of cookies the same origin policy. Cookies are small strings sent by a server to the browser. The browser holds onto these strings, sending them back to the server in the headers of subsequent requests.

The cookie same origin policy is related, but distinct from the JavaScript same origin policy. The cookie policy refers to how cookies are stored and sent back to the server [94, 95]. While any server can set a cookie in the browser (subject to additional constraints imposed by the browser), each cookie is associated with a specific domain name. Only requests to a server with the same domain name as associated with the cookie will receive that cookie when a request is made by the browser. Additional restrictions can further restrict the sending of cookies to a domain depending on SSL state, port number, or path name of the specific request.

The *HTTP-only* cookie extension prevents JavaScript from accessing cookies which are sent between the client and server, even if the JavaScript has come from the same origin as the cookie [111].

SOMA operates at the granularity of requests, restricting fetches for remote content. If a request is blocked by SOMA, all headers related to the request, including the cookie header, will not be sent to the remote server.

Flash Same Origin Policy

The same origin policy dictates that JavaScript running in a web browser is not allowed to read content which comes from a different origin than the base HTML page. Scripts embedded into Flash are not subject to the restrictions imposed by browser, but instead those imposed by Flash [117].

The Flash same origin policy is very similar to the same origin policy implemented in the web browser, with one exception: for a Flash object coming from origin A, requests by the Flash script can be made to origin B if and only if B provides a `/crossdomain.xml` file listing origin A. This allows a script to perform cross-domain requests (in contrast to JavaScript AJAX requests – see Section 3.1.1).

3.7.2 Alternative Methods of Improving JavaScript Security

NoScript [105, 106] involves the user maintaining a white-list of sites the user authorizes to run JavaScript. The ability of NoScript to protect the user is directly related to the user's ability to maintain the white-list. Indeed, many recent versions of NoScript have introduced additional components more in line with SOMA. These components do not require user intervention and can render harmless certain clickjacking and cross-site request forgeries – SOMA provides a consistent approach for blocking both (and indeed more web vulnerabilities as discussed above). NoScript, with its white list, is not backwards compatible, breaking many existing web sites. We believe that in its current form, NoScript is unlikely to be enabled by default by any large browser vendor.

Web-based execution environments have all been built with the understanding that unfettered remote code execution is extremely dangerous. SSL and TLS can protect communication privacy, integrity, and authenticity, while code signing [142, 149] can prevent the execution of unauthorized code. Neither approach protects against the execution of malicious code within the browser. Java [46] was the first web execution environment to employ an execution sandbox [179] and restrictions on initiating network connections [46]. The Java policy for restricting network communication was designed to prevent Java applets from communicating to any server other than the one it was retrieved from.

Subsequent systems for executing code within a browser, including JavaScript, have largely followed the model as originally embodied in Java applets.

While there has been considerable work on language-based and module-based sandboxing [83, 60], only recently have researchers begun addressing the limitations of sandboxing with respect to JavaScript applications.

3.7.3 Alternate Protections Against Specific Web Attacks

There have been many attempts in the past to protect against various web-based attacks. Some, including a proposal related to SOMA by Schuh [147] involve the browser enforcing firewall-style rulesets provided by the origin as a way of protecting against several different attacks. Other approaches focus primarily on a single type of web attack. These approaches include defences against cross-site scripting and cross-site request forgery as described below.

Cross-Site Scripting

Cross-site scripting vulnerabilities exist when the parse tree for a page can be modified in a way not intended by the web application developer. By restricting changes to the parse tree generated by the browser, the opportunity exists to eliminate cross-site scripting vulnerabilities. Ter Louw et al. propose Blueprint [103], an approach for generating the parse tree for the web page at the server and ensuring it is not interpreted differently at the client. Their approach, while covering many of the cross-site scripting vulnerabilities not protected against by SOMA, requires all web application developers to buy in to the approach, redesigning their applications to match the requirements of Blueprint.

Another approach that can be implemented on the server involves performing dynamic taint tracking (combined with static analysis) to detect the information flows associated with XSS attacks [178]. Noxes [89] is a client-side web proxy approach which uses manually and automatically generated rules to mitigate possible cross-site scripting attacks.

Barth et al. [15] proposed a defence against cross-site scripting attacks performed through content sniffing. Their solution involves modifying browsers to restrict content sniffing such that executable JavaScript can not easily be embedded into content uploaded to the web server.

The Mozilla content security policy [158] (previously called site security policy [159]) focuses primarily on mitigating cross-site scripting attacks. It is much more complex a policy than SOMA and requires modifications to the web applications themselves. SOMA does not require the same buy-in from web application developers.

Bojinov et al. [25] introduced a multi-service variant of cross-site scripting, along with a potential solution. The variant involves injecting the malicious content through a non-web based vulnerability. Such an approach may be feasible on devices which host multiple services (e.g., inject a web attack via the ftp service). If the injected attack initiates a bootstrap (as discussed in Section 3.6.4), SOMA prevents the bootstrap.

Cross-Site Request Forgery

Several approaches have focused on mitigating cross-site request forgery. Barth et al. [16] proposed a solution to the login based cross-site request forgery using the HTTP `Origin` header and web application based firewall rules on the server. Jovanovic et al. [78] proposed a server-side proxy solution where a session ID is embedded into every link found in the web page generated by the server. Unless the subsequent request from the client browser contains the unique session ID, it is rejected by the proxy (before it can be processed by the real web server). Web applications written to defend against cross-site request forgery attacks are likely to use the standard approach of themselves embedding the session ID into each URL sent out by the web server [16], negating the need for the proxy.

The SOMA approach of using approvals also prevents cross-site request forgeries from being sent to the web server, but does not require a proxy. Instead, the browser is responsible for preventing such requests from being sent.

3.7.4 Mashups

Recently several researchers have focused on the problem of web mashups, which may be created on the client or server. Client-side mashups are composite JavaScript-based web pages that draw functionality and content from multiple sources. To make these mashups work within the confines of same origin policy, remote content must either be separated into different iframes or all code must be loaded into the same execution context. The former solution is, in general, too restrictive while the latter is too permissive; client-side mashup solutions are designed to bridge this gap. Two pioneering works in this space are Subspace [74] and MashupOS [68, 181]. SOMA restricts communication between the web page (browser) and servers while mashup solutions restrict local communication between elements on the page.

SOMA breaks client-side mashups which use code hosted on a site not included in the manifest. In order for a mashup to work with SOMA, every web site involved must be explicitly listed in the manifest and also allow its content to be included (through responding YES to an approvals request). While such restrictions may inhibit the creation of new, third party mashup applications, they also prevent attackers from creating malicious mashups (e.g., combinations of a legitimate bank's login page and a malicious login box). SOMA is designed such that it can be implemented on sites that wish the increased protection that SOMA provides. Mashup sites may choose not to enable SOMA. SOMA does not affect server-side mashups.

3.7.5 DNS Rebinding Attacks

DNS rebinding attacks are one method of bypassing the current same origin policy [71, 72]. The attack involves rapidly changing the IP address associated with a domain name so that multiple unrelated servers are associated with the same domain name, allowing JavaScript read and modification of content across the different physical servers. Karlof et al. [79] proposed a solution to this which involves tying the origin of a page to the X.509 certificate instead of the DNS name.

SOMA operates in the browser using and blocking HTTP requests. It relies on the web browser for host name to IP address resolution. SOMA is therefore susceptible to the same DNS rebinding attacks as the browser itself. The protection against DNS rebinding attacks, when implemented in the browser, will also provide protection to SOMA. One such solution is DNS pinning [71], where the browser forces all content for a domain to be fetched from the same IP address. When applied to SOMA, DNS pinning would result in all content being fetched form the same IP address as the related SOMA manifest or approval request.

3.7.6 Restricting Information Flows

While the general problem of unauthorized information flow is a classic problem in computer security [49], little attention has been paid in the research community to the problems of unauthorized cross-domain information flow in web applications beyond the structures of same origin policy — this, despite the fact that cross-site scripting and cross-site request forgery attacks very heavily rely upon such unauthorized flows. Of course, the web was originally designed to make it easy to embed content from arbitrary sources. With SOMA, we are simply advocating that any such inclusions should be explicitly approved by both parties.

While SOMA is a novel proposal, we based the design of `soma-approval` and `soma-manifest` on existing systems. The `soma-approval` mechanism was inspired by the `crossdomain.xml` [3] mechanism of Flash. External content may be included in Flash applications only from servers with a `crossdomain.xml` file [3] that lists the Flash applications' originating server. Because the response logic behind a `soma-approval` request can be arbitrarily complex, we have chosen to specify that it be a server-side script rather than an XML file that must be parsed by a web browser.

The `soma-manifest` file was inspired by Tahoma [40], an experimental VM-based system for securing web applications. Tahoma allows users to download virtual machine images from arbitrary servers. To prevent these virtual machines from contacting unauthorized servers (e.g., when a virtual machine has been compromised), Tahoma requires every VM image to include a manifest specifying what remote sites that VM may communicate with.

Note that individually, Flash's `crossdomain.xml` and Tahoma's server manifest do not provide the type of protection provided by SOMA. With Flash, a malicious content provider can always specify a `crossdomain.xml` file that would allow a compromised Flash program to send sensitive information to the attacker. With Tahoma, a malicious origin server can specify a manifest that would cause a user's browser to send data to an arbitrary web site, thus causing a denial-of-service attack or worse. By including both mechanisms, we address the limitations of each.

3.8 Final Remarks

It is interesting to note that most of the attacks prevented by SOMA can already be mitigated by web developers properly using existing security mechanisms. The fact that web application vulnerabilities are so prevalent is a testament to the inability for web application developers to handle the complexities of designing a secure web application. SOMA is designed to greatly simplify the development of web applications by providing a run-time environment to the web developer which provides greater isolation and hence a greater level of security. SOMA focuses on those operations which are inherently dangerous and seeks to limit them. The approach discussed in this chapter does not rely on the end-user for enforcement, is enforced on all web applications using the SOMA enabled browser, and is implemented by guardians (the browser vendor and web server administrator). It therefore follows the thesis objective of providing a guardian based mandatory access control mechanism which can be deployed.

4 Limiting Privileged Processor Permission

In this chapter, we look at a mandatory access control policy mechanism which already exists but which has not been fully utilized in many systems – separating root and kernel-level control. We argue for the increased use of this policy mechanism as a method for protecting the kernel (both in Linux and Windows). We examine the protection of the kernel against software running at lower protection levels, including applications and scripts which execute on a typical system. For the purposes of this chapter, we focus on the protection levels common in a typical desktop computer.

4.1 Background

While the focus of this chapter is on applications running with root level privileges and their ability to modify other elements on a desktop, as background we first review the different protection levels which exist within a modern desktop.

4.1.1 A Permission Hierarchy

The modern desktop computer system is composed of a number of different protection level layers that have been designed to segregate different elements of the system [157]. These protection levels are illustrated in Figure 4.1. Within each stack (software, or any system device), all the higher protection level elements can read and write to areas occupied by the elements at the lower protection levels. We now discuss each of the protection levels as well as what aspect of the desktop system is responsible for maintaining the separation between the protection levels. We do not examine the possibility of malicious hardware [88] in this thesis.

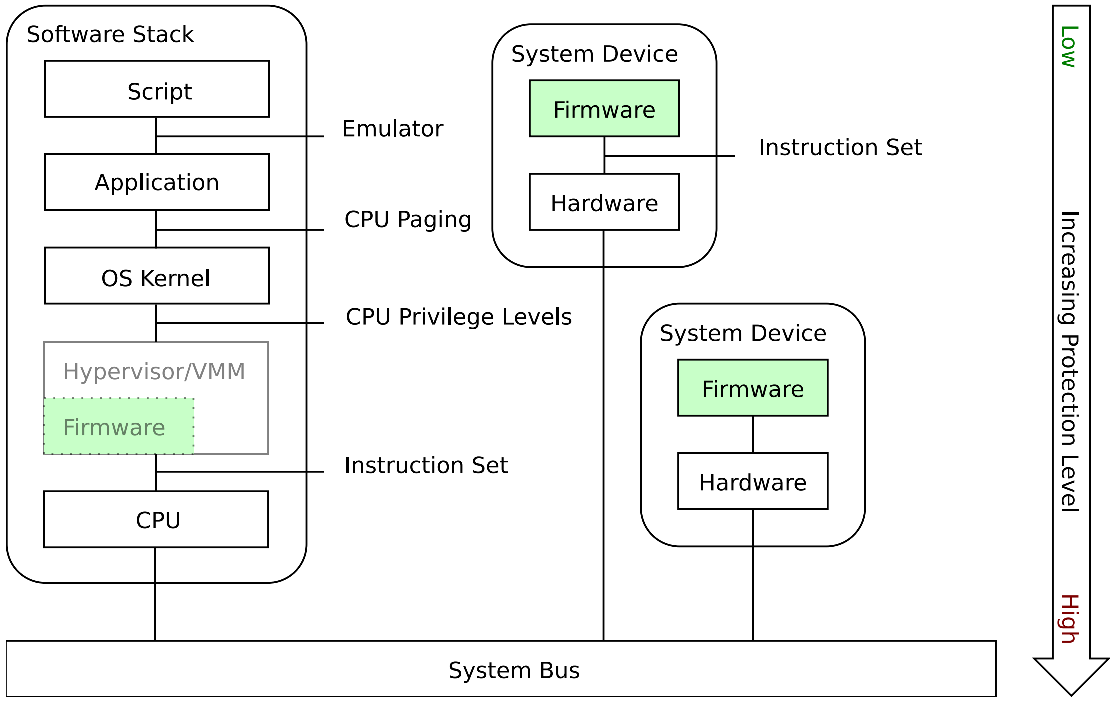


Figure 4.1. Enforcement between protection levels in a modern desktop.

A System Device

Typically, system devices are composed of hardware and firmware working together to provide functionality. The hardware hooks into the system bus and exposes the functionality provided by the device (e.g., for an optical drive, this allows the software stack to access certain properties of the drive as well as the data on any inserted optical disk).

The privileges of the device firmware, including its ability to interact with the rest of the system, are dictated by the underlying hardware (both that of the device the firmware is run on, as well as hardware the device is connected to). These restrictions can be in the form of specific hardware limits imposed to prevent damage or limits in the form of functionality which is simply not made available to the firmware. As an example, the range of motion for the read head for an optical drive is limited by the firmware, while the ability to prevent writing to optical media may be prevented by simply installing a laser not powerful enough to actually 'burn' media.

Many devices are built in such a way that the firmware can be upgraded. For these devices, the new firmware must transition from the software stack to the system device. If either the CPU or hardware on the system device disallows the upgrading of firmware, the upgrade will fail. A similar situation occurs for those devices that do not store their own firmware in non-volatile memory but instead rely on the software to re-upload the firmware during system initialization. This situation is similar to the upgrading of firmware, but happens on a more frequent basis. In Figure 4.1, we illustrate the firmware that gets sent to the system device as being contained within the hypervisor layer. While the firmware can in fact also be housed at any lower protection layer in the software stack (and be subject to the privilege restrictions of all the higher protection levels as it transitions onto the system device), firmware modified and stored at any lower protection level may present a way of bypassing the protection mechanisms imposed on the lower protection levels. Firmware malcode exploits this ability to circumvent the protection levels within a desktop [65], and indeed methods of protecting against firmware malware have also already been proposed [139, 2].

The Software Stack

The most visible protection level stack that exists on a common desktop is the software stack. As illustrated in Figure 4.1, it includes the processor, potentially a hypervisor, operating system kernel, applications, and scripts. At the base of the software stack is the CPU, the processor that all the software runs on. The processor is responsible for many of the protection mechanisms used to keep other elements in the software stack at their respective protection levels. It does this through a combination of two mechanisms, the paging subsystem and the privilege level subsystem [153].

The paging subsystem presents a translation layer between the virtual addresses used by software running on the processor and the physical addresses corresponding to the actual location in memory that is being referenced. Each block of virtual memory can be mapped to an arbitrary block of physical memory, with the exact mapping being maintained as an entry in the page table. The processor restricts the ability to update the page table to only privileged code (i.e., the OS kernel). The OS kernel is responsible for maintaining this mapping (for the moment, we will assume a hypervisor is not present). When an application is being run, the page table mappings are configured to allow access only to physical memory allocated to the currently running application. Because the mapping is controlled by the operating system, the application is restricted from accessing memory belonging to other applications on the system. While the OS kernel shares the same page table as the application, memory associated with the kernel remains protected through a privilege level bit enforced by the processor.

In its simplest form, the privilege level subsystem of a modern processor is a single bit, indicating whether the currently executing instructions have *privileged* or *user* level control of the processor [43]. Privileged mode is normally associated with the operating system kernel, with user mode being associated with all other code which is running. This separation between privileged and user mode is not the same as is commonly referred to in many access control systems (more on this later). In addition to some assembly instructions only being accessible to code running with privileged control, the paging subsystem is capable of restricting access to memory based on whether the code currently running is privileged. A specific page of memory can be indicated as read-only for any user code running but read/write for privileged code. The OS kernel uses this privileged bit and associated access restrictions in the page tables to prevent user code from writing to pages belonging to the operating system (including memory pages containing the page tables themselves). Any attempt by user code to either execute privileged assembly instructions or modify read-only pages of memory is trapped by the processor and forwarded to the privileged code. The privileged code can either reject the attempt or emulate the operation on behalf of the user code.

This setup, with the OS kernel being privileged and all other code being restricted in the operations it can perform, provides the basis for the software protection level stack. When a hypervisor is inserted into the mix, it becomes the code to which privileged permission is given by the processor and the OS kernel is instead run with user processor protection. Any attempt by the OS kernel to then execute privileged operations can be caught by the hypervisor and handled accordingly. The hypervisor can also ensure that the OS kernel is kept separated from other code running with user permission.

For scripts that execute in the software stack, it is the job of the script interpreter to impose any desired restrictions on the script. The script interpreter, in turn, is restricted by the operating system, and so on down the chain. A good example of this is JavaScript running in a browser, which is prevented from accessing the local file-system even though the browser itself can access files (subject to constraints imposed by the OS kernel). The same holds true for Java, where the byte code is interpreted and controlled by the JVM, which in turn is an application controlled by the OS kernel.

4.1.2 Methods of Modifying Kernel Code

On a modern desktop system, almost all code is run with user level processor control, regardless of the access control mechanisms being imposed (indeed, it is the job of the OS kernel to impose any additional access control restrictions required for the system). Every single application, even those run by root, is still run as user code as far as the processor is concerned.

Because all applications are run with user level processor control, they do not have the same privileges as those given by the processor to the OS kernel. Instead, they need to request that certain operations be done by the OS kernel on their behalf (so that they are allowed by the processor). The OS kernel itself typically restricts the interfaces that are available for changing itself (and hence allowing new code to run with privileged processor permission). Typically, the OS kernel will provide a few interfaces for extending its code.

Physical Memory Access

The kernel may export to applications an interface which allows arbitrary access to the physical memory of the machine [173, 172, 143, 148]. This allows an application to:

1. Talk to hardware mapped into memory.
2. Read from and write to memory allocated to another application already running on the system.
3. Read from and write to memory allocated to the kernel.

Because this interface allows arbitrary modifications to all code currently residing in memory, permission to use this interface is normally restricted by the OS kernel to only applications running with the highest privileges (superuser). On Linux, this interface is typically a device node labelled as `/dev/mem`. Another related device node which only exposes kernel memory for reading and writing is `/dev/kmem`. On Windows, the device node for modifying physical memory is `\Device\PhysicalMemory` [41].

Kernel Modules

A second way of expanding the OS kernel with new code that is considered privileged is through loading a kernel module [143, 104]. Kernel modules allow new code to be inserted into the kernel and run as privileged by the processor. This provides a structured stable way of expanding the functionality of the OS kernel (as opposed to modifying the kernel through physical memory access, which tends to be fragile).

Swap

In a modern kernel, the swap (or page) area on disk is designated for excess memory allocated by a process which is not currently being stored in physical memory [157]. The contents of physical memory are written (or paged) to disk and the physical memory reassigned by the OS kernel.

A related kernel interface provides applications with the ability to read from and write directly to storage, bypassing the file-system and associated access controls. Performing such reads and writes is typically referred to as performing *raw disk I/O*. When the ability to write to storage is combined with the kernel paging to disk, applications inherit the ability to modify memory potentially associated with other applications. If portions of kernel memory are paged out to disk, applications also can modify the kernel though writing to the area of disk occupied by swap [145].

Kernel Image

When a system first boots, the kernel must be loaded from somewhere before it can start running. If an application has the ability to write to the area of disk occupied by the kernel, the application can modify the code which is run with privileged processor permission after the next reboot of the system. To protect the kernel, we must ensure that the kernel image on disk can not be updated by malware. We must also ensure that the boot loader (e.g., GRUB [81]), which is responsible for loading the kernel, cannot be modified by malware. In this thesis, we do not consider the case where the physical media is inserted into an alternate machine and then the kernel is updated.

Replacing the running kernel image is possible in Linux through the `kexec` system call [133]. This functionality allows the booting of arbitrary code on an already-running system, giving it privileged processor control. To protect the kernel, we must also disable the running of arbitrary code through `kexec`.

4.2 The Protection Mechanism

To date, root or supervisor level control of a system has been closely associated with kernel level control. They are, in fact, not the same. All applications run as user and the kernel runs as privileged as far as the processor is concerned. The kernel is also in charge of enforcing the protections associated with root and other user accounts, while not having to follow them itself. Many kernel level rootkits have taken advantage of the extra power that comes from being part of the OS kernel and running with privileged processor control.

There exists, however, an opportunity to keep root and privileged CPU control distinct. Because there are only a few ways of elevating one's privileges between the two (as discussed in Section 4.1.2), one can concentrate on these select areas to increase the protection afforded to privileged code, protecting the kernel against threats by root level applications. To enforce the protection mechanism, one only needs to implement additional restrictions on the limited number of methods available for escalating from root application level control to privileged processor control. We do this by restricting access to those methods which are discussed in Section 4.1.2. We document these restrictions in Sections 4.2.1 through 4.2.5. We believe that protecting the kernel in this way is beneficial. Malware exploitation of the kernel is a growing trend [12] and others in the security community have already made an effort to protect the kernel in complementary ways (we discuss such approaches in Section 4.6).

The protection mechanisms discussed in this chapter must all be implemented on a system in order to prevent code from gaining privileged processor permission. The mechanisms, however, do not need to be deployed across all installs of Linux in order for the benefits of protecting the kernel to be realized. The approach discussed in this thesis is incrementally deployable (at the granularity of a system).

4.2.1 Restricting Memory Access

In restricting access to privileged processor code, the first step we discuss is disabling write access to memory occupied by the OS kernel. On Linux, this involves restricting write access to `/dev/mem` and `/dev/kmem`. Restrictions to these device nodes have already been implemented in Linux by others, and need only be enabled. As of Linux version 2.6.26 [173], the option exists to limit access through `/dev/mem` to only those areas of physical memory associated with IO (e.g., the graphics card). All areas of the physical address space associated with RAM can not be written to through `/dev/mem` when the option is enabled. The ability to disable access to kernel memory through the `/dev/kmem` device node has also been configurable since 2.6.26 [172]. Before being introduced in the mainline Linux kernel, the options had been used in Fedora and other Red Hat kernels for 4 years without any known problems [173]. `/dev/mem` cannot be disabled entirely as X (the graphical display manager typically used in Linux) uses it to communicate with the video card.

In Windows, the writes directly to physical memory are accomplished by using the `\Device\PhysicalMemory` device [41]. This device, however, has been disabled by Microsoft since Windows 2003 SP1 [143, 110].

4.2.2 Restricting Access to Disk

Because the OS kernel is responsible for mediating all access to the underlying hardware on a system, it has the ability to control access to the underlying disks. By restricting access to those areas of the disk being used by swap, writes to swap can be prevented (and hence the integrity of kernel memory can be protected). To fully prevent against the kernel being modified, the areas of disk occupied by kernel modules, the core kernel, and the boot loader also need to be protected against arbitrary modification. Arbitrary writes to sectors of the disk occupied by any of these elements need to be restricted (i.e., a root level process must not be allowed to perform arbitrary writes to either `/dev/hda` or `/dev/hda1` if they contain kernel-related elements).

Raw Disk Access

In restricting raw disk access, we implement a simple protection rule which is sufficient to protect against all of the kernel elements being modified. Any partition actively being used by the OS kernel (either mounted or being used as swap) can not be written to via the raw interface. This includes both partitions used as swap as well as any partition with a file-system that is currently mounted. We focus on raw writes in this thesis, not restricting raw read operations.

The disabling of raw writes to partitions that have been mounted has a side benefit. For those partitions that have been mounted, the kernel maintains a cache of recently used disk blocks to increase efficiency. Writes to the underlying disk for any block being cached by the OS are not guaranteed to be preserved, leading to potential corruption of the file-system.

While Linux currently does not implement such restrictions, such a restriction is beneficial for both from the stability and security standpoint. In our prototype, we extend the kernel to implement such a restriction. Microsoft's Windows Vista implements similar restrictions on accessing raw disk blocks [116]. Because the boot loader in general often exists outside the file-system that is exported to user-space applications, the act of disabling writes to the raw disk has the additional advantage of protecting the boot loader.

File Access

If raw disk writes are disabled, the only way left of modifying the swap file, kernel, modules, or boot loader on disk is through the standard open, read, write, and close system calls associated with files. Access control on these file-system operations is enforced by the OS kernel. To protect the swap file, the kernel need only prevent an application from opening and writing to the swap file. In Linux, the ability to modify the swap file is not normally separate from the ability to modify other files on the system. To protect the swap file, the kernel needs to be extended to treat the swap file as special and disallow write attempts, even those done by root. Windows Vista already employs such an approach to prevent modifications to the page file [143].

The core kernel and related modules on disk are also commonly available as files on the file-system. While the naïve approach to protecting such elements would be to disallow all updates to these files, such a solution does not allow upgrades. To allow upgrades, an approach such as bin-locking discussed in Chapter 5 can be used. Because the kernel and related module files are all binaries on disk, the bin-locking approach is well suited to protecting them against arbitrary modification. The boot loader can also be protected by treating the area of disk occupied by the boot loader as a binary file and using bin-locking.

4.2.3 Kernel Modules

To prevent new arbitrary code from executing on a system with privileged processor control, one must also restrict the loading of kernel modules. The commonly accepted way of restricting these is through the use of kernel module signing. While not currently accepted into the mainline Linux kernel, a patch does exist to implement kernel module signing [96]. It enforces that only modules signed with the private key corresponding to the public key embedded in the OS kernel can be loaded to extend the kernel. The signature is contained within an ELF [166]. The public key is embedded into the core OS kernel during compile time. Only someone in possession of the private key can create a kernel module which verifies and is loaded into the running kernel. Windows Vista introduced a similar approach, preventing arbitrary kernel modules from being loaded [143, 96].

4.2.4 Updating the Kernel Image

While only an issue when the machine is rebooted, updating of the kernel image is nevertheless still an avenue through which the OS kernel can be modified. To prevent modifications to the OS kernel, an approach like bin-locking (introduced in Chapter 5) can be used to restrict updating the kernel image. We discuss such an approach above in Section 4.2.2.

To prevent updating the running kernel image through `kexec`, we must limit the kernels that are allowed to be started by calling the `kexec` system call. In limiting the kernels allowed to be run through `kexec`, we prevent arbitrary code from running with privileged processor control. In current (unmodified) kernels, `kexec` is by default disabled, and set by the compile option `CONFIG_KEXEC`. Furthermore, the option is only available for X86 processors.

4.2.5 Hardware

While certain hardware configurations may provide additional methods of gaining write access to kernel memory, we believe that the OS kernel device drivers responsible for the underlying hardware can be modified to remove vulnerabilities caused by specific hardware. This may involve restricting both the interface and operations exposed to applications. This requires further exploration, but is beyond the scope of this thesis. The exact method for ensuring hardware does not allow additional code to run with privileged processor control is highly dependent on the exact hardware being examined.

4.3 Benefits

Many have decided to abandon hope in securing the kernel from within, declaring it instead as intrinsically insecure and electing to use the hypervisor level to implement security mechanisms designed to protect the kernel (see Section 4.6). The hypervisor is also used to provide isolation between the kernel and underlying hardware. The aim of isolating the underlying hardware from the operating system has yielded benefits in virtualization and server consolidation, allowing one physical machine to run multiple operating system instances simultaneously. The aim of protecting the operating system against attack by applications using virtualization, however, to date has only been discussed in academic circles (to our knowledge).

In the spirit of a guardian enforced mandatory access control policy, we recall the execute disable flag introduced to combat code injection buffer overflow attacks (discussed in Section 2.6.1). The solution worked not because it was the most complete (it was not, it only worked at a page level and only protected against code injection buffer overflow attacks), but because it was simple and widely deployed. Instead of depending on an entirely new protection level to protect the kernel, we focus on cutting off the obvious methods through a few simple mechanisms, forcing malware to exploit a vulnerability (which is much harder) in order to gain access. Any of the more complete methods discussed in Section 4.6 can be used in combination with the protection mechanisms we propose in Section 4.2 for a defence in depth approach.

4.3.1 Detection versus Prevention

In enforcing the separation between root and privileged processor control on a system, we prevent many current kernel level rootkits from working on a system. While there are many actions that can be taken once a kernel rootkit has privileged processor permission [12, 67, 77], the ways of getting that permission are very few.

The other approach that can be taken toward addressing kernel rootkits is to attempt to detect them either as they access the interfaces to gain privileged processor control, or once they are in the kernel. Detection, however, is a reactive protection method. We believe that when possible, it is much better to prevent the threat from occurring rather than to attempt to detect it. In our case, the ability for rootkits to gain privileged processor control can be prevented, and hence detection approaches may not be necessary. Prevention approaches work best when the interface is not commonly used by legitimate software (which in this case is true, as discussed in Section 4.5). We discuss the many methods that have been proposed to detect against privileged processor control rootkits in Section 4.6.

4.3.2 Restriction of Raw Disk Writes

The additional protection mechanism most likely to affect end users (or the applications they are likely to run) is that of disallowing raw writes to partitions being actively used by the OS kernel (including both swap and file-system partitions). This protection mechanism is necessary to prevent applications from bypassing the standard file access controls which have been imposed by the OS kernel. It also prevents swap from being written to as a method for modifying code run with privileged processor permission.

There are a number activities that are performed on all systems using the raw disk interface provided by the OS kernel. We now discuss each of these and how they are affected by disallowing raw writes.

1. **Disk Partitioning** - Disk partitioning involves allocating areas of a disk for use by different file-systems. Creating and modifying disk partitions requires write access to the underlying raw device by the partitioning software. This operation is inherently dangerous, and can be very destructive to data on the partitions. Regardless of the protection mechanisms proposed in this thesis, modifying the partition table of a drive which is currently mounted is still very dangerous. We therefore see restricting the modification of partitions on drives which are mounted as beneficial.

2. **Partition Formatting** - Formatting involves creating a new file-system on a partition created as discussed above. Creating a new file-system on a partition which is currently being used by the OS kernel is never recommended, being likely to lead to both corruption and data loss, either in swap (if the partition is currently being used for swap space), or on the file-system which was mounted at the time of the format operation. Again, regardless of the additional protection mechanisms presented in this thesis, formatting a mounted partition is a very dangerous.

3. **File-System Checks** - A file system check involves checking the consistency of a file-system, ensuring it is free of errors. As long as a file-system check does not modify an active partition, the consistency check can be considered a relatively safe operation (and indeed would be allowed by the new protection mechanism, since writes would not be performed). File-system checks that write to a mounted partition are inherently dangerous and should not be performed. Current approaches for checking a mounted file-system involve requesting that all modifications be performed by the kernel, to avoid file-system corruption [108].

In all cases above, the ability to perform raw disk access for writing combined with the fact that the partition accessed is also being used by the kernel leads to a dangerous scenario. The ability to restrict raw disk writes to mounted partitions is beneficial, regardless of whether malware is involved. Indeed, users who try to perform a file-system check on a mounted partition often end up with corrupted data. We view the addition of a protection mechanism to addresses these dangerous activities as advantageous.

4.4 Limitations

By further limiting the interface between user and privileged processor control, additional restrictions are introduced where previously developers were unrestricted. The restrictions, however, are being placed on aspects of the system very seldom (if ever) used by an end-user or the applications they are likely to run. Kernel debuggers, one application broken by the restrictions on being able to write to privileged kernel memory, are very unlikely to be run by end users. For the small subset of users who **must** use kernel debuggers or modify the raw file-system without rebooting, the protection mechanisms discussed in this chapter are not appropriate. Most end users, however, do not need to use kernel debuggers. End-users are also accustomed to having to reboot when partitioning disks, formatting, or repairing file-systems.

4.4.1 Raw Disk Access

While the inability to perform raw disk access may be viewed as a limitation of our system, we actually view it as a benefit as discussed in Section 4.3.2.

4.5 A Prototype Implementation

To test the feasibility of better enforcing the protection barrier between root level and privileged processor control, we implemented the protections discussed in Section 4.2. This included preventing raw disk access on partitions that were mounted, disallowing kernel module loading, and restricting access to physical memory interfaces. Our prototype implementation used Debian 4.0 for applications and Linux kernel version 2.6.25 modified to enforce the additional protection mechanisms.

The boot process was modified on the test system to initialize kernel data structures that limit raw writes (to both mounted partitions and the swap partition). For memory access, `/dev/kmem` was disabled and `/dev/mem` was restricted to only allow writing to areas of physical memory not occupied by RAM. In an alternate prototype, which we discuss in detail in Chapter 6, we test preventing modification of the kernel or associated modules on disk.

4.5.1 Restricting memory access

In the prototype Linux kernel, we enabled the pre-existing kernel options for restricting access to `/dev/kmem` [172] and limiting the memory writable through `/dev/mem` [173]. We disabled the running of arbitrary new kernels by omitting the `kexec` system call (`CONFIG_KEXEC` was set to 'n' in the kernel configuration).

4.5.2 Restricting Kernel Module Loading

In the prototype Linux kernel, we disabled kernel module loading, since our goal was in determining whether everyday user activities would be influenced by the increased kernel protection mechanisms. An alternate approach is to restrict kernel module loading based on signatures, as already discussed and implemented by Kroah-Hartman [96].

4.5.3 Disabling Raw Disk Access

To test the restriction of raw writes to mounted partitions, we modified the prototype Linux kernel. We export a new syscontrol from the modified kernel, allowing a user space process to set which partitions should prevent (disable) raw disk writes. A syscontrol is a single pseudo-file (a file which does not exist on disk) that exposes kernel configuration to user space. In this case, the list of protected partitions can be read by user-space applications, and a new partition can be appended to the list by writing to the pseudo-file. Because the syscontrol only supports appending to the list maintained by the kernel, the only way to remove a partition from the list is to reboot the system, clearing the list. As part of the boot-up process, the list of partitions for which raw disk access is disabled is written back into the syscontrol (after the initial `fsck`/file-system check). This list of partitions written to the syscontrol in the prototype included the swap partition (to prevent attacks against kernel memory [145]). If any partition on a disk is being protected, the prototype kernel also disables raw writes to the file representing the entire drive. In order for malware to enable raw disk writes, it must modify the start-up process to disable initialization of the syscontrol and reboot the system. While we discuss protecting the startup process in Chapters 5 and 6, we note that this avenue for attack is specific to our prototype implementation, not a general problem with disabling raw disk writes.

In the implementation, the restriction on raw disk writes was implemented as a user-specified list. As an improvement, the file-system code in the Linux kernel could be modified to automatically prevent raw writes as the partition is mounted.

4.5.4 Protection Against Current Rootkits

To verify that the prototype system was able to defend against OS kernel rootkit malware, we attempted to install several Linux rootkits.[1]. We selected two kernel-based rootkits (`suckit2` and `mood-nt`), attempting to install them on the system using the provided install programs. Both failed to install because of disabled write access to `/dev/kmem`. The fact that both rootkits depended on `/dev/kmem` gives weight to disabling access to kernel memory. We believe additional mechanisms proposed in this chapter raise the bar significantly for an attacker attempting to compromise the kernel.

The prototype discussed in this chapter was implemented as a component of

[1]All Linux rootkits tested were from `http://packetstormsecurity.org/UNIX/penetration/rootkits/`

the $\sim$ 2000 line bin-locking kernel module discussed in 5.6. The performance overhead the kernel module, including bin-locking, is discussed more in Section 5.6.4.

4.5.5 Effect on Applications

During development and use of the prototype (including watching videos, listening to music, browsing the web, reading e-mail, and writing a paper [192]), we did not encounter any programs that were blocked by the other kernel interface restrictions implemented in the prototype. We did not encounter any program (other than malware) which attempted to write directly to swap or kernel memory.

4.6 Related Work

We now discuss related work.

4.6.1 SELinux Reference Policy

We start our discussion by examining the SELinux reference policy[2] and its method for protecting against each of the methods discussed in Section 4.1.2. We note that because SELinux is a default-deny policy system, much of the policy itself focuses on granting permissions, rather than documenting why a permission is not granted. While many of the methods for gaining privileged processor control are denied by the SELinux reference policy, it is because these permissions are not required in order for applications to run correctly.

1. **Physical Memory Access** - In restricting access to kernel memory, the reference policy groups together under the same permission access to the device nodes `/dev/mergemem`, `/dev/oldmem`, `/dev/kmem`, `/dev/mem`, and `/dev/port`. Applications such as `kudzu` (which detects and configures new and/or changed hardware on the system), `vbetool` (which communicates with the video BIOS), and `xserver` (which provides the graphical interface in Linux) are all granted write access to these device nodes. Of the three, `kudzu` has been deprecated by Red Hat [136], and the other two

[2]We examined version 2.20090730, available from `http://oss.tresys.com/projects/refpolicy/wiki/DownloadRelease`

use the privilege to access the video card, leading to the possibility that the SELinux reference policy can be made finer-grained by focusing on providing access to just the video card. The current status of each device node is as follows:

- `/dev/kmem` - Provides access to kernel memory. Enabled by the kernel configuration option `CONFIG_DEVKMEM` (see Section 4.2.1), and enabled in the standard Debian (v5.0) build of the kernel.
- `/dev/mem` - Provides access to all system memory. Restricted to portions of the memory address space not associated with RAM by enabling the kernel option `CONFIG_NONPROMISC_DEVMEM` (which was renamed to `CONFIG_STRICT_DEVMEM` in kernel 2.6.27). This kernel option is not set in Debian v5.0.
- `/dev/mergemem` - Used for combining identical physical pages of memory in an effort to reduce memory consumption. While the documentation of the Linux kernel was updated in 2.1.115 to include this device node, the related code appears to never have been accepted into the main Linux kernel tree. Mergemem does not appear to have been actively maintained since January 1999.
- `/dev/oldmem` - Used by crashdump kernels to access the memory of the kernel that crashed. Useful for debugging the kernel, and can be disabled on production machines. Enabled by the `CONFIG_CRASH_DUMP` configuration option, but disabled on the standard Debian (v5.0) build of the kernel.
- `/dev/port` - Provides access to I/O ports. Enabled by the kernel configuration option `CONFIG_DEVPORT`, and enabled in the standard Debian (v5.0) build of the kernel.

With physical memory access, the SELinux reference policy does not make the distinction between kernels that have been compiled with the raw memory options discussed in Section 4.2.1, and those that have not. Because SELinux operates at the object level, it relies on the objects themselves being specific enough to limit permission correctly. Without restrictions on where data can be written using `/dev/mem`, providing write access to video memory has the side effect of providing write access to kernel memory.

2. **Kernel Modules** - The ability to load kernel modules is granted to a single program in the SELinux reference policy; `insmod`. The ability to run `insmod` is then restricted, in effect preventing arbitrary kernel modules

from being loaded by restricting what subjects are allowed to execute the `insmod` binary. In the reference policy, subjects with the `bootloader_t` type are allowed to execute `insmod`.

3. **Swap** - In the SELinux reference policy, the ability to access swap if it exists as a physical partition on the hard drive is limited to those programs allowed to perform raw writes to fixed disks. Of all the methods of obtaining privileged processor control, being allowed to perform raw writes to the hard drive is the only one documented in the reference policy as allowing SELinux security protections to be bypassed.

   If swap is a file instead of a partition, there are no specific rules in the reference policy for restricting write access to the active swap file. We suspect it is up to the individual deploying SELinux to ensure that the swap file can not be written to.

4. **Kernel Image** - The ability to replace the kernel image on disk (and, indeed, the bootloader or kernel modules) is determined through a number of different actions. To prevent the updating of the raw disk blocks corresponding to these files, the SELinux reference policy has rules for restricting raw disk access (as discussed in point 3). To prevent updating these files at the file-system layer, the SELinux reference policy restricts write access to the bootloader, kernel image, and kernel modules separately. A subject given write access to the bootloader may not have necessarily have write access to the kernel modules. Replacing the running kernel image by using `kexec` is not mentioned in the SELinux reference policy.

Within the SELinux reference policy, there is no overreaching privilege that, if given to a process, allows that process to obtain privileged processor control through *some* method. In following with the general SELinux philosophy of least privilege, the developers of the reference policy have attempted to restrict, for each object and subject, the actions that subject can perform on the object, without making it clear that allowing these actions allows arbitrary additional privileges because the subject can gain privileged processor control.

4.6.2 Microsoft Windows

Microsoft's Windows Vista already implements many of the kernel access protections discussed in this chapter, including restrictions on write access to raw drive partitions, swap, and kernel memory [116, 114, 145]. The success of such approaches in the Windows environment provides strong evidence that such restrictions are not inherently detrimental to typical system use. Our discussion focuses on providing a complete view of the required protection mechanisms that must be implemented to protect the kernel from malware. We provide similar protection in the Linux environment, which has traditionally provided a more open interface to developers. We also build on some of the mechanisms provided by Windows, proposing a method for selectively allowing updates to the kernel and module files on disk. We believe the mechanisms discussed in this chapter provide important new protection within the Linux kernel while still allowing it to be open (any individual can still compile and use their own kernel). The approach taken in this chapter relies on bin-locking (Chapter 5), but can also be modified to rely on `configd` (Chapter 6). Both bin-locking and `configd` rely on a secure kernel, which the work in this chapter supports.

4.6.3 Other Related Work

By far, the most common approach for detecting OS kernel level malware is to have a detection mechanism installed at a different protection level (recall Figure 4.1). Copilot, discussed by Petroni et al. [131] operates as a distinct system device on the system. It monitors the OS kernel in an attempt to detect changes to static kernel elements such as privileged processor code. It also provides a mechanism for partial restoration of changes made by malicious kernel rootkits. Petroni et al. [132] likewise use a system device to detect changes in the OS kernel, but concentrate on protecting dynamic data structures.

Wang et al. [182] discuss an approach that attempts to detect software that exists for the purpose of hiding certain resources on a system (i.e., ghostware). It compares the results of examining certain aspects of a system from two different angles to determine if there are any differences. These snapshots of system state are taken at the same time. Their approach resides at the application privilege level and examines the results returned by the high level OS kernel API to specific queries. A second set of requests performed to the low level OS kernel API is then queried and compared to the first to determine if there are any discrepancies. Like CoPilot above, this technique detects ghostware after it has infected the OS kernel, not preventing the infection.

Kruegel et al. [98] present an approach for detecting at load time whether a kernel module is malicious through binary analysis. This approach is the most similar to the protection mechanisms proposed in this chapter in that it attempts to prevent the loading of malicious code. Running in the OS kernel, it also relies on the protections against swap and physical memory writes as discussed in Section 4.5. This approach can be used instead of kernel module signing as a method to protect the running OS kernel against the insertion of additional code which runs with privileged processor permission.

Carbone et al. [32] take a snapshot of memory allocated to the running kernel and attempt to map all dynamic data contained within it. Using the memory snapshot and the corresponding kernel source code, they create a directed graph of memory usage within the snapshot. They then check function pointers and detect hidden objects as a method for detecting kernel rootkits. The approach operates offline, and works to detect rather than prevent rootkits.

Several approaches [59, 132, 183, 152, 52, 11] have leveraged recent advances in virtual machine monitors (VMMs) to detect kernel malware. These approaches leverage the VMM as a way of protecting the detection mechanism against malware which has managed to compromise the OS kernel while still allowing complete analysis of the lower protection levels of the software stack.

Because the interface between the OS kernel and VMM is similar to that between the VMM and underlying hardware, higher level knowledge about OS kernel structures must be reconstructed from what is seen at the hardware layer. Many approaches refer to this as *introspection* – rebuilding the state of the OS kernel at the VMM layer in order to obtain a higher layer understanding. Because work involving VMMs assumes that it is not possible to secure the OS kernel, they are forced to examine the OS kernel from a higher protection level. We believe that the protections discussed in this chapter show that it is possible to separate the OS kernel from root level processes, re-establishing trust in the kernel. In trusting the kernel, we allow many of the protection mechanisms developed to be moved from the VMM back into the OS kernel, making introspection unnecessary.

One argument for declaring the OS kernel insecure is that there are many bugs that allow applications to compromise the kernel. While it is true that the number of lines of code in a modern kernel is high, we do not believe the discovery of bugs to be sufficient justification for declaring the kernel unsecurable. Indeed, research on finding and fixing software vulnerabilities through static and dynamic analysis [55] presents an opportunity for reducing these vulnerabilities. As hypervisors become more complex, it is increasingly likely their security will be equally affected by bugs.

In trusted computing platforms such as AEGIS [9], the kernel's digital signature is verified by the boot-loader before it is loaded. Because the kernel currently exports interfaces which allow it to be updated by user-space applications, a cryptographic hash of the kernel at boot time is insufficient to verify the integrity of the currently running kernel. In order to verify that the running kernel has not been modified, the integrity verification must include all applications which have ever run on the system since boot [146]. We believe such an approach is overly heavyweight compared to disabling privileged processor access on end-user desktops.

4.7 Final Remarks

Many of the elements presented in this chapter for restricting access to the kernel have previously been proposed individually in some form or another. The combination of all these elements, however, provides what we believe to be complete protection against root level processes being able to obtain privileged processor control. Protecting the kernel against compromise is a goal shared by many other researchers, as evidenced by the large volume of research published on the topic. This chapter takes the approach of protecting the kernel through prevention, rather than detection. In taking the avenue of preventing access to the kernel, we parallel the work of execute disable, using processor features to prevent, rather than try to detect, an attack. The approach discussed in this chapter does not rely on the end-user for enforcement, is enforced on all applications deployed for the system, and is implemented by a guardian (the OS developer). It therefore follows the thesis goal of providing a guardian enforced mandatory access control mechanism which can be deployed.

5 Bin-Locking: Selectively Restricting Updates to Binaries

In this chapter, we describe a file-system protection mechanism designed to limit modifications to *binaries* – library and executable files on disk. We focus on establishing a beachhead against malware by protecting binary files – a type of file very rarely modified by a user. Our proposal, however, extends to other types of files not modified by the user of a system.

5.1 Background

In current computing environments, software applications written by many different authors all coexist on disk, being installed at various times by the user. Each application normally includes a number of program binaries along with some associated libraries. While the installation of a new application will normally not overwrite previously installed binaries, permission to modify all binaries is common. Indeed, this raises a problem: *Any application installer (or even application) running with sufficient privileges can modify any other application on disk.* Application installers are routinely given these privileges during software upgrade or install (i.e., almost all installers run as administrator or root, giving complete access to the system). Some applications even run with administrator privileges during normal operation due to a variety of reasons and despite the best efforts and countless recommendations against this practice over the years. The frequent running of applications (including their installers) as administrator leads to a situation in which a single application can modify any other application binary on disk. Normally, applications do not abuse this privilege to modify the binaries belonging to other applications. Malware, however, does not traditionally respect the implicit expectations followed by normal software, and uses the ability to modify other binaries as a convenient installation vector. This situation is not new. Already in 1986, the Virdem virus [154] was infecting executables in order to spread itself; more recently, rootkits [51] have used binary modification to help escape detection.

5.2 The Protection Mechanism

Our approach is to associate a digital signature with each library file and executable which is checked by the kernel when an update to the binary is attempted by software. No centralized (or other) public key infrastructure is involved, although an important design aspect is a relationship between the signature on the new binary, and a public key embedded in the (old) binary being replaced. We use additional kernel protections to restrict modification of these signed binaries on a system. The core technology is based on a simple application of code-signing or self-signed executables [189]. We use the term *bin-locking* (short for binary-locking) to refer to our proposal, to avoid confusion with other code signing mechanisms. The proposed system allows binaries to be transparently and securely upgraded, facilitating the application of security patches.

At the core of our proposal is a simple but well-planned use of digital signatures, designed to protect a binary against unauthorized modifications. To restrict who can modify (or replace) a binary on disk, we enforce one simple protection rule: *A library file or executable on disk can only be replaced by a library or executable containing a digital signature verifiable using any public key in the previously installed binary having the same file name.* Binaries protected by the bin-locking proposal have embedded within them a set of digital signatures along with a set of corresponding public keys.[1] We propose supporting a few standardized digital signature algorithms (the particular choice is specified alongside and protected by the digital signature). The kernel, upon finding a digital signature (in the bin-locking section of the binary), will restrict replacement of the binary to those new binaries containing a digital signature which can be verified using any key in the currently installed binary. If the signature can be verified (or the currently installed binary is not bin-locked), then the replacement is allowed. Otherwise, the replacement is denied by the kernel and the original binary remains unmodified on disk. Only one signature needs to be verified in order for the replacement operation to succeed. Deployment is incremental – binaries not bin-locked can be replaced without

[1]The portion of the file holding the digital signature itself is not included in the range of the signature, to prevent a recursive definition. The public key, however, is integrity-protected by the signature.

restriction (which is how most systems currently operate), but once a binary is bin-locked it must be replaced by a bin-locked binary. The proposed method is quite different than SDSI/SPKI [54, 141]. We trust keys only in a very limited setting (for replacing a binary which was signed with the same key). We do not use names or certificates.

Over time, inevitably, signing keys used for bin-locking will be lost or compromised. Both situations can be ameliorated by allowing, as an option, the embedding of multiple verification public keys in a binary file. If one corresponding private key is lost, the other(s) can be used to sign a subsequent version of the file (which can also introduce new keys). We do not specify any conditions on who or what controls the private keys corresponding to these additional verification public keys, but many options exist including community trusted organizations or trusted friends who function as backups. While we mandate no specific infrastructure for key revocation, pro-actively installing a new version of a file which does not allow future versions signed with the previous key (i.e., which excludes the old verification public key(s) from those embedded in the new version) prevents a compromised key from being a threat indefinitely. Because each file can be signed with a different key, the effect of a compromised key can be limited. When or how frequently to change keys which are not known to have been compromised is a question we do not focus on in this thesis.

5.2.1 Trust Model

In the proposed system, it is assumed that a malware author does not have physical access to the end-user machine and that malware does not have access to the private signing keys, nor have kernel level control of the target system.[2] The protection mechanism discussed in Chapter 4 is suitable for protecting the kernel.

5.2.2 A Generic Approach

At its base, the bin-locking approach restricts updates to an object based on the signature verification public keys embedded in the object. In our discussion thus far, we have restricted objects to be binary files because they are not

[2]We define the kernel (and hence kernel level control) to include only those aspects running with elevated CPU privileges (ring 0 privileges on x86) – this definition of kernel does not include core system libraries installed alongside the operating system but run in user space.

commonly modified by users and have a defined structure. The bin-locking approach, however, can be expanded to any type of object not modified by end users. This includes data files used by an application (e.g., fonts and graphics). Objects can also be complete application packages (including all the files contained within, but not user data files created with the application) – this is the approach taken by Android [5], discussed in Section 5.7.1.

5.2.3 Interdependence on Signing Keys

The parts of the binary file signed in creating a bin-locked file include all structure surrounding the signatures, including the bin-locking section headers, public key prefixes, public keys, and any other element except the actual digital signature. Because the data used to generate each digital signature includes all public keys contained in the file, a single public key or signature cannot be replaced in a bin-locked file signed by multiple parties without other parties having to re-sign the file. This approach is necessary because a single matching signature is considered sufficient in updating a binary file. If a bin-locked file could be signed with an additional signature while still being legitimate for those keys already embedded, an attacker could take control of a binary file by appending a new signature to an already bin-locked file.

5.2.4 Kernel Modifications

To ensure that bin-locked files remain visible on the file-system, we must ensure that a new file-system is not mounted over top of bin-locked files, and that a file-system containing bin-locked files is not unmounted unexpectedly. We first recognize that the mounting and unmounting of file-systems is not commonly performed (or at least does not affect core system directories) after system boot. We therefore modify the kernel to prevent mounting and unmounting of file-systems on specific paths. In the prototype, the list of such paths to protect is fully customizable by the user or machine administrator, being set as part of the boot process (however once a path is specified, it cannot be removed from the list of specified paths). We discuss the specifics of the prototype in Section 5.6.2.

To rule out an easy way to subvert the proposed protection mechanism, we must also disable raw disk access to those partitions containing bin-locked binary files. This was discussed in Chapter 4.

If deleting application binaries were still allowed, the bin-locking system would be rendered ineffective; the attacker could simply delete the binary and then install a new one. Therefore, in a system supporting bin-locking, signed application binaries may not be deleted. To delete bin-locked files in the proposed system, the bin-locking protections must be disabled. In the prototype, this requires a reboot into a kernel which does not enforce the protection mechanism (see Section 5.6.6). Previous work on providing a trusted interface (e.g., see [193]) – one which cannot be subverted by malware – between the kernel and the user may help to eliminate the reboot requirement. One solution (not implemented in the prototype) is to tie enforcement of bin-locking to whether or not a hardware token is inserted (similar to a mechanism proposed by Butler et al. [30]) – as long as the hardware token is inserted, moves and deletes to signed binary files would be allowed.

5.3 Benefits

We believe previous proposals which attempt to limit changes to binaries on disk all fall short for one of three reasons: they either detect changes after they have happened [87, 113, 8, 174] (making recovery hard), rely on the user to correctly validate every file modification (imposing usability issues), or still allow applications to modify any file at all during install/upgrade [174]. Bin-locking addresses these three points and yields additional benefits as explained in Sections 5.3.1 to 5.3.5.

5.3.1 No Central Key Repository or Infrastructure

The proposed system differs from other code-signing systems currently in use in that it does not attempt to tie the signature to an entity. It can verify that the new version of an application binary was created by the same author (or organization) as the old version without knowing who the author is. Because the signature on a to-be-installed binary file is verified using the public key embedded in the previous version of the same file (by file name) installed on the system, there is no need to centrally register a key or involve any central repository. Thus, *no central certification authority or public key infrastructure (PKI) is required.* An application author can create a signing key-pair and begin using it immediately. Furthermore, if desired, different keys can be used for each file on a system, limiting the impact of a key compromise (as long as all private keys are not stored in one place the attacker gains access to, the attacker incurs a per-executable cost for replacing protected binaries). Because there is no dependence on a centralized trusted authority, development of new software remains unrestricted. We make no effort to restrict the software which can be installed on a system, as long as that software does not modify already-installed binaries. Other digital signature schemes proposed in the past have relied on a trusted central authority [8, 174, 50].

5.3.2 Trusted Software Base Even After Compromise

The legitimate operating system and core applications are normally installed before malware attempts to infect a machine. We exploit this temporal property. Typical malware, because it is installed after the operating system (including core libraries and programs), is not capable (if the proposed system is in place for operating system files) of changing any of the operating system files. The operating system files, therefore, can be trusted to be unmodified. If an anti-virus system is installed before any malware, the anti-virus binaries can also be automatically protected using the same mechanism. Core binaries on a compromised system can therefore be trusted, allowing much greater control over an infected system without requiring a reboot to clean media. Furthermore, this ability to reliably trust binaries on the system can make recovery easier [62]. Using the proposed system, anti-virus software can be protected against modification and system binaries can be relied upon with confidence in their integrity, restricting the ability for malware to modify or filter the results of such applications in an attempt to hide. In the case of forensic analysis, while administrators may still choose to reboot to known-clean media once they discover malware, the inability for malware to hide is likely to result in the administrator becoming aware of the problem earlier.

5.3.3 Incremental Deployability with Incremental Benefit

Kernels which do not yet support the bin-locking mechanism will treat signed binary files as normal binary files, since the modifications made to a binary to support bin-locking are backwards compatible. Similarly, binaries without bin-locking digital signatures are allowed on a kernel which supports the proposal (in contrast to most proposed code signing schemes [174]). Either the kernel or libraries can be updated to support bin-locking first without an adverse affect on non-supporting systems.

5.3.4 Low overhead

As will be discussed in Section 5.6.4, the prototype has performance impact which is imperceptible to the end-user.

5.3.5 Simplicity

The bin-locking proposal is relatively simple to understand and does not require any additional hardware or co-processors. Because it is simple, developers are more likely to understand the protection mechanism. Also, the user is not involved in enforcing the protection mechanism, eliminating the risk of the security mechanism failing due to end-user error.

5.4 Limitations

We now discuss some limitations related to deploying the proposed system. We note that many of these deficiencies can be addressed by combining the bin-locking approach with `configd`, as discussed in Chapter 6.

5.4.1 Denial of Service

Any attempt to install an application after malware is already on the system and has written bin-locked binaries (with different signatures) to the file names required by the application will result in a failed install. While this “limitation” may be viewed as an advantage from a security viewpoint, not all users will find that this improves usability. Currently, many users continue to use a computer after it has been infected with malware. They either are not aware that malware is there, or are not aware of the full implications of malware being on the system. Malware initiating such a denial of service attack will illicit a forced user response when an application install is prevented. At this point in time, the desire of the user to install an application is a security benefit in encouraging them to clean their system and remove the malware, including the files which resulted in the denial of service. We note that in order to take advantage of their desire to install an application, the process of removing the malware (including the reboot required in the prototype system) must be as simple as possible; the usability impact of this makes the proposal more suitable for some user environments than others, for example, for expert users, or users supported by technical experts.

The same user response is forced by malware performing a denial of service through filling the hard drive with bin-locked files which cannot be deleted (without a reboot). Both malware actions result in a state where the user (or their technical support team) is aware and forced to take action on installed malware. For malware that wishes to hide, it seems unlikely that either denial of service will be actively exploited.

5.4.2 Signed Binary Moves and Deletes

With the bin-locking system in place, file deletion and movement become much more complex. We cannot allow a bin-locked file to be easily moved or deleted since this would open up a method for allowing file replacement. We note, however, that application (and operating system) binaries seem to be rarely moved or deleted on a system. Furthermore, we believe (and personal experience seems to support) that users rarely uninstall applications. Statistics from the Debian popularity contest project indicate that for many applications, a large percentage of the people who have installed the application have not used it within the past month [129] (e.g., 95.358% of people who have installed `tuxkart` have not used it in over a month according to data collected on March 6, 2010). For those times where an uninstall or move is required, a reboot into a different kernel would allow the operations to be performed. Again, we acknowledge that the usability impact of this makes the proposal in its present form more suitable for some user environments than others. We discuss reboots as they relate to the prototype implementation in Section 5.6.6.

5.4.3 Aliases

The proposed protection mechanism only protects binaries on disk. If malware can prevent the correct binary on disk from being invoked, then it may still take precedence over legitimate programs. As an example, running the `ps` command from the prompt without a pre-pended path (i.e., fully qualified file name) will cause the first copy of `ps` found to be run (even though it may not be the `/bin/ps` binary). While the bin-locking scheme is designed primarily to protect binaries against modification, bin-locked binaries provide no additional protection if they are not invoked. We must ensure therefore on an infected system that the legitimate binary can be easily run instead of one found at a location of the attacker's choice. Additional copies of binaries installed by malware can be avoided by running bin-locked applications directly, avoiding environment variables such as `PATH`. There are a number of methods for accomplishing this, including calling the kernel directly (e.g. using the `execve` system call) to run a program. Because much of the aliasing functionality is implemented by libraries likely to be protected by the bin-locking scheme, some aliasing vulnerabilities can be avoided (e.g., by restricting `PATH` to include only core system directories when running as root).

5.4.4 Developer Systems

While we want developers to use bin-locking for the applications they build, developer systems are not the design target of the developed applications. We discuss several developer features which currently remain enabled on all systems and undermine the security of bin-locking. We propose disabling these features on systems using bin-locking (e.g., end-user desktops) in order to increase security. We believe that typical end-users (as opposed to developers) will not be bothered by the following restrictions.

1. `Ptrace` hooks [128] are used by developers to debug a running application. By allowing reads and writes to a process memory space, arbitrary changes to both data and code within the running application can be made. In order to ensure bin-locked binaries are run unmodified, ptrace access needs to be disabled for bin-locked binaries.

2. The customization of binaries by third parties is made much more difficult by bin-locking. The modification of binaries by third parties, however, is exactly the type of attack that bin-locking aims to prevent. Bin-locking ensures the software run by end-users is never modified by anyone other than those who developed the software. Developers wishing to switch between original software and custom builds (e.g., during debugging) will not be able to take advantage of the benefits of bin-locking for those binaries.

3. Preloaders on Linux such as `LD_PRELOAD` allow additional libraries not specified in an executable to be linked in at run-time. The use of preloaders thus provides a method for modifying a binary at run-time. There are two ways of preventing this from being exploited by attackers. The first (and easiest) is to disable `LD_PRELOAD` on non-developer machines (e.g., by installing a bin-locked `/lib/ld-linux.so.2` not implementing the feature). The second defence against `LD_PRELOAD` comes through recognizing that it must be processed (indirectly) by the binary during start-up. If the `LD_PRELOAD` environment variable is ignored or reset to a known-

good value (e.g., the empty string), the attack vector is disabled. While most applications leave the functionality intact (i.e., by calling the default `/lib/ld-linux.so.2`), the application developer can disable the functionality on binaries they intend to bin-lock.

5.5 Extensions to the Core Approach

Assuming a kernel supporting bin-locking has the basic capability of verifying that binary updates are authorized, other extended functionality may be worth considering. While we did not implement the following extensions in the prototype, we believe them to be useful extensions to the core idea; those discussed here would require additional support from the kernel.

5.5.1 Versioning

As one possible extension to the system, version numbers could be embedded in both the old and new binaries. If the kernel limits replacement based on version number, the same public-private key pair could be used over an extended period of time without the risk of a downgrade attack (i.e., replacing a more recent binary with an older version containing a vulnerability). While authors (or organizations) can achieve the same effect by "revoking" keys (as discussed at the top of Section 5.2), versioning allows the software author to minimize the number of public keys which must be contained in any new version of the binary while still ensuring that the binary can replace many previous versions. We acknowledge that rollback by legitimate users (the process of reverting software to a previous version) may not be possible while bin-locking enforcement is active on a system (since bin-locking, as outlined in this thesis, is designed to also prevent downgrade attacks).

5.5.2 Sub-Keying

In the core idea, any binary which can be validated using signature verification public keys in the installed binary will be allowed to replace the installed binary. To prevent one binary from replacing another binary with different functionality written by the same organization (preventing their software from working properly), the organization could use a non-overlapping set of keys for each binary. As an extension to the basic bin-locking idea, an organization could embed an index number into each binary they sign. New versions of the same binary would have the same index number; binaries for different applications developed by the same organization would have different index numbers. If the kernel enforces that the index number between the old and new binary must match, an organization could use the same private signing key for all their binaries (without allowing their binaries to be maliciously switched on a system). As an example, sub-keying could be used to prevent the contents of `rm` from being used in an update to `ls` while allowing the two binaries to be signed with the same key.

5.5.3 All Key Verification

The core approach of bin-locking states that a library file or executable on disk can only be replaced by a library or executable containing a digital signature verifiable using *a* public key in the previously installed binary having the same file name. Only one signature needs to be verified in order for the replacement operation to succeed.

An alternate approach is to require that either all signatures in the new version of the binary be verifiable using appropriate keys in the old version, or alternatively, that all keys in the old binary be used in the verification of the new binary (i.e., for each public key in the old binary, there is a corresponding valid signature in the new binary). While similar, the two approaches are different. We now discuss each.

Variation 1: Verifying All Signatures in the New Binary

In the base proposal, as long as at least one signature matches, we can conclude that the new version of the binary came from a trusted source. Variation 1 requires that all signatures present in the new binary verify using public verification keys in the old binary. Variation 1 is not much different from verifying a single signature from the perspective of restricting updates. If the enforcement policy is that all signatures present must verify in the new version, the developer of the binary could simply omit all but one signature, reducing the approach to the base single signature verification scheme (leaving only one signature that can be verified using a public key in the already-installed binary).

In variation 1, the developer needs to be able to add a public verification key without adding a corresponding signature to a new binary (i.e., signatures can not be tied directly to keys, as done by `jarsigner` [125]). Because any signature in the new version of a binary must have a corresponding public verification key in the old version, adding a new signature when any new key is added would result in a new binary which fails the verification check of variation 1. Our proposal uses key prefixes to allow the introduction of public verification keys without the introduction of associated signatures.

Variation 2: Verifying Using All Keys in the Old Binary

Variation 2 requires that for each public key embedded in the old version of a binary, there is a corresponding verifiable signature in the new version of the binary. This variation prevents a new version of the binary from being installed unless *all* groups holding private keys corresponding to the public keys in the installed binary sign the new binary. This approach can be used to prevent one developer from "going rogue" – taking over the application by distributing a new version only signed with their private key (even though the developer attempting to go rogue has a public key contained in the already installed version of the binary). This variation prevents a single compromised private key on a binary signed by multiple parties from being usable – the attacker would have to compromise all keys in order to replace the binary. The downside to this approach is that the non-malicious loss of even a single key (e.g., through hard drive failure) would result in no updates to the binary being possible.

Similar to variation 1 above, variation 2 requires that new public verification keys can be added to the binary without adding an associated signature. If a signature can not be embedded into the binary without also embedding the corresponding public verification key, a key could not be phased out of new versions of the binary while still having the variation 2 check pass. If version n is signed with private keys A, B, and C, then all future versions $> n$ must also be signed with private keys A, B, and C unless signatures can be added without adding the associated verification public key. Our proposal uses key prefixes to allow the introduction of signatures without the introduction of associated public verification keys.

5.5.4 Security Updates by Regular Users

The ability to restrict updates to only those files that are bin-locked with a verifiable signature presents another opportunity. On many systems, the user of the system is not the same individual as the administrator (especially in business and educational environments). By allowing updates to bin-locked applications, we can safely allow users to install patches without granting them administrator access. Bin-locking provides the mechanism to ensure a user only updates bin-locked binaries with new valid bin-locked binaries. Of course, allowing any user to update a binary with a new version is not appropriate in all environments. We therefore propose still using traditional file access control policies to dictate which users have permission to update a file in the first place. In the prototype below, we continued to enforce traditional access control policies (such as Unix discretionary file-system access controls) in addition to those of the bin-locking system.

5.6 A Prototype Implementation

To verify the viability of the bin-locking proposal, we modified a Debian 4.0 Linux system to implement bin-locking, including the kernel interface restrictions discussed in Chapter 4. The prototype implementation is composed of a number of different pieces which together protect the system. We wrote a binary signing utility which is used along with associated custom scripts to sign the binaries in the Debian software archive (for Debian 4.0), creating a new local Debian mirror which we used for testing. We then installed these binaries on a test system using the Debian package manager, which we modified to support bin-locked binaries. The Linux kernel (version 2.6.25) on the test system was modified to enforce the proposed protection mechanisms (which include restrictions on bin-locked binaries as well as access to the kernel and file-systems). The boot process was modified on the test system to initialize kernel data structures which limit raw writes and mounting. We discuss each of these steps in detail below.

5.6.1 Extensions to the ELF Format

Executable files for a particular operating system normally follow a standard structure. Most Unix distributions (including Linux) use the binary format file ELF (Executable and Linkable Format). The basic ELF file is represented in Figure 5.1. Except for the ELF file header, all other elements are free to be arranged as desired. We modified ELF files (our approach could be adapted to other types of files not modified by the user – e.g., Windows executables, Windows libraries, or application data files), creating a new type of ELF section for storing bin-locking related data. The ELF binary file format was designed such that applications could create new sections and many other applications (e.g., GCC and bsign) take advantage of this flexibility.

The new bin-locking section of the ELF file is made up of one or more records (the section table contains a field specifying the number of records), each containing a type of digital signature (e.g., all elements related to the DSA algo-

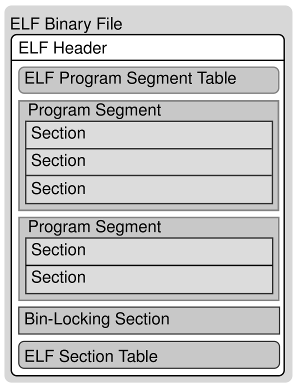


Figure 5.1. Basic ELF layout including bin-locking section.

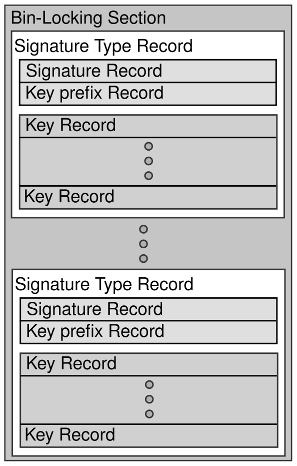


Figure 5.2. Bin-locking file section layout

rithm would be in one record). Each record specifies a signing algorithm (type), signature, prefix of the public keys that can be used to verify the digital signature, and zero or more keys that are eligible to verify digital signatures of the same type in subsequent versions of the binary. The key prefix record contains the first four bytes of the public verification key related to the digital signature and is used for quickly determining what verification key in a previous version of the binary should be used for verifying the digital signature (if multiple keys share the first four bytes, the kernel will attempt to verify with each). We illustrate the layout of the bin-locking section in Figure 5.2.

To allow for future signature schemes, we included several additional variables and flags in bin-locking section headers. The header for records and sub-records was specified to include both a length and type field, allowing the modified kernel to skip over unrecognized signature types. The prototype's sub-record header, in addition, contains a flag that signals the kernel to zero the data part of the record when hashing the file; signatures are stored in sub-records with this flag set. We discuss the kernel verification of digital signatures more in Section 5.6.2. The layout of a sub-record is illustrated in Figure 5.3.

5.6.2 Kernel Modifications

The kernel was modified to enforce bin-locking as discussed in Section 5.2. Consequently, the modified kernel does not allow signed binary files to be deleted, moved, or opened for writing. They can only be replaced with new binary files

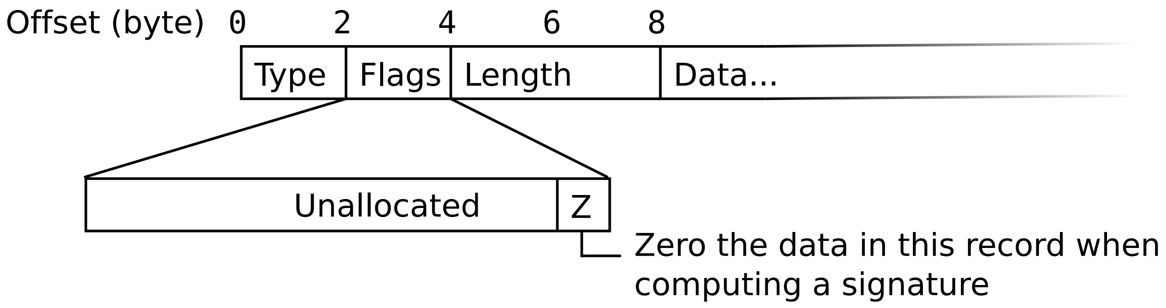


Figure 5.3. Layout of a sub-record in the bin-locking section (records are similar).

that contain a signature verifiable using keys in the old binary. On a replacement request (which is initiated through a `move` kernel system call involving a bin-locked binary), the kernel attempts to extract the keys from the old binary and use them to verify the validity of the new binary. If the signature in the new binary successfully verifies, the kernel moves the new binary over top of the old binary. Using the move system call, however, presents a dilemma. The new binary must not be bin-locked (or the move will be denied), but at the same time it must be bin-locked (to replace the old binary). The prototype kernel deals with this by ignoring the first eight bytes of the new binary when performing movement validation checks (a feature used by the modified Debian package manager discussed in Section 5.6.3). The addition of eight new bytes at the beginning of the ELF file change it such that the kernel does not recognize it as bin-locked (for the purposes of preventing its movement and deletion) but still recognizes it as bin-locked (allowing it to replace another bin-locked file if the signature verifies). During the replacement, the kernel will first verify the signature on the new binary (ignoring the first eight bytes). If signature validation passes, the kernel will move the new binary over top of the old one, removing the initial eight bytes during the move (the file is locked to prevent it from being modified between signature validation and the move operation). The bin-locked binary resulting after the move is protected by the kernel. Because binaries currently being run cannot be modified on disk (in any Linux system), a move must be used (instead of a copy or other update mechanism) to replace bin-locked binaries. The prototype kernel retains backwards compatibility with binary files that are not bin-locked, not restricting their replacement or removal. Currently, the prototype kernel can verify application binaries signed using the *Digital Signature Algorithm* (DSA) [56].

We chose not to rely on a user space helper in designing the prototype system in order to simplify the implementation. While we believe a user space helper could be implemented securely, it would need to be bin-locked itself, and the interfaces it uses to talk to the kernel would have to be designed very carefully to avoid being susceptible to being taken over by malware. Even with keeping the entire implementation in the kernel, the number of new lines of code added to the kernel was around 2000.

Detecting Bin-Locked Files

To detect whether or not a file is bin-locked, the modified kernel examines very specific elements in the file. It verifies that the file: 1) is an ELF file [166], 2) contains a bin-locking section, and 3) in the bin-locking section there exists a digital signature in a format known to the kernel (currently, only DSA signatures are supported). To determine if a file is bin-locked, the kernel must read the file header (the first 52 bytes of the file on a 32bit x86 platform) as well as the section table (40 bytes per section). If any element in either the file header or section header is considered invalid according to the ELF specification [166], the file is treated as not bin-locked. An attacker is not able to turn a bin-locked file into one not bin-locked because the kernel does not allow a properly bin-locked file to be altered (except by replacing it with another properly bin-locked file). The eight byte header described above results in a file that is not recognized as a valid ELF file and hence the modified kernel allows it to be removed, modified, and moved. We assume individuals attempting to bin-lock their own binaries will not purposely create invalid binaries (as that would negate the effort of bin-locking the binary in the first place). As part of the prototype, we created a tool to test that properly signed ELF files are recognized as valid. The overhead of enforcing bin-locking, is imperceptible to end-users of the running system (a Pentium 4 at 2.8GHz with 1G of RAM), as discussed in Section 5.6.4.

Verifying Digital Signatures

To verify that a new binary can replace an already existing bin-locked binary, the modified kernel first extracts out a list of verification keys from the old binary (which shares the same prefix as the key prefix stored in the new version of the binary alongside the digital signature). Using each verification key that matches the key prefix, the kernel attempts to verify the digital signature. If the verification passes, the replacement is allowed. If the binary contains two different types of digital signatures, the modified kernel will allow replacement if any one of them contains a verifiable digital signature.

While the prototype uses an older and less efficient implementation (space-wise) to store and verify digital signatures than described herein, the method proposed in this sub-section is preferred.

Avenues for Kernel Modification

To protect the bin-locking system itself, the kernel was also modified to remove known functionality which could be used to attack the system. The main such known attacks are 1) modifying the kernel to disable the protection scheme, 2) editing the bin-locked binaries directly on disk, and 3) hiding bin-locked binaries (by either mounting or unmounting the partitions they reside on). To prevent 1), we used kernel protection mechanisms discussed in Chapter 4. While we disabled module loading entirely, a better option is to deploy module-signing (as discussed by Kroah-Hartman [96]). We also limited raw disk access and drive mounting (as discussed below).

Disabling Raw Disk Access

To protect bin-locked binaries against modification, one must also disable raw writes for partitions that contain bin-locked files (on both mounted and unmounted partitions). We did this by using the syscontrol from Section 4.5.3. As part of the boot-up process, the list of partitions for which raw disk access is disabled is written back into the syscontrol (after the initial `fsck`/file-system check). In order for malware to enable raw disk writes, it must modify the start-up process to disable initialization of the syscontrol and reboot the system. One solution to prevent this is to initialize the syscontrol in `init` (the first binary run). Binaries involved in the start-up process (including `init`) can be bin-locked, preventing modification.

In the implementation, the restriction on raw disk writes was implemented as a user-specified list because the kernel could not determine quickly what partitions contain bin-locked files. As an alternative, the file-system could be modified to include a flag indicating the presence of bin-locked files on that partition. If bin-locked files are present, then raw writes to the partition could be automatically disabled without the kernel needing a list. By avoiding file-system modifications, the prototype was able to operate at the security module layer [187], not depending on a particular file-system. By leaving the file-system unmodified, backward compatibility with systems not aware of bin-locking is also maintained.

Restricting Mounting

To prevent bin-locked files from becoming inaccessible to user-space applications, the prototype restricts the locations where file-systems can be both mounted and unmounted using the same approach of a syscontrol which supports both read and write operations. By writing “`< /usr/lib`” to a syscontrol created by the prototype, the modified kernel enforces that no file-system can be mounted or unmounted at `/usr/lib`, `/usr`, or `/`, meaning that all files in `/usr/lib` continue to be accessible until the system is rebooted. By writing "`> /usr/lib`" to the syscontrol, no file-system can be mounted on any subdirectory or parent directory of `/usr/lib` (i.e., > implies <). File-system root rotations are also not permitted by the prototype if the syscontrol restricting mounts has been written to. Although both the new syscontrols support write operations, all writes to these syscontrols are converted to appends by the modified kernel and hence cannot be used to modify previous entries written to the bin-locking related syscontrols. Remounting partitions to enable and disable write access must be allowed, as this functionality is used during the normal shutdown process to avoid file-system corruption.

We currently see no easy method of avoiding the list of mount location restrictions. Administrators may require unmounting on devices containing bin-locked files (e.g., unmounting removable media). While it is possible to prevent mounting a new file-system over bin-locked files using a file-system flag (as discussed above), whether to prevent file-system unmounts depends on the environment.

5.6.3 Modifications to Executable Files

To bin-lock binary files, the prototype used binary rewriting. An application was created which would use one or more signing keys to sign an existing binary, injecting into the binary both the signatures and all the verification public keys related to the signature (we chose not to use bsign [29], preferring a simpler method). ELF files are used for both program executables and shared libraries – the bin-locking approach covers both. Currently, the signing application signs binaries using DSA, although this can be extended to other signature formats. The modification of executables is backwards compatible. Signed (i.e., bin-locked) binaries can be used seamlessly on a system which does not understand bin-locking.

We modified the Debian package manager [47] to not write out bin-locked binaries to temporary files during the installation of the system (since once written into a temporary file, the modified kernel will not allow the file to be moved or deleted). Instead, the package manager writes out an eight byte prefix (we used the prefix `CODESIGN`) followed by the signed file (which is recognized as such by the modified kernel during replacement of the original signed file, as mentioned in Section 5.6.2). The binary rewriting application was used along with several additional scripts to create a local Debian 4.0 mirror [31] where every application binary and library was bin-locked.

One element in the standard Debian boot process initially posed an issue for our bin-locking process. During the boot process, the temporary initial RAM disk (a file-system within RAM which stores files used early in the boot process) is deleted because it is no longer necessary. If this initial RAM disk contains binaries that are bin-locked, the new kernel prevents the delete. To overcome this, bin-locking is disabled in the prototype on drives not associated with a physical device.

As partial evidence that the modified kernel and signed executables are viable, a paper [192] was written on a test system (which used the prototype implementation). On the system, all binaries and libraries were bin-locked and kernel restrictions were active. The test system, while running KDE (the graphical based K Desktop Environment) was also used to browse the web, write e-mail, listen to music, and view video – all with no noticeable differences from an ordinary system. This confirmed that it is possible to lock down the kernel interface, as well as use bin-locking on a deployed system, with the resulting system still usable for everyday tasks.

5.6.4 Performance

Any performance impact of the proposed system was imperceptible to this thesis' author, the end-user, during the writing of a paper [192] (and indeed while performing other common activities such as web browsing, video watching, and image editing). For more precise measurement, we ran benchmark tests to quantify the overhead of the system. Using the Perl benchmark library [156], we measured the average increase in kernel time required to perform a delete and move operation on both non-ELF and unsigned ELF files with an ext3 file-system. Over 25000 test runs (10000 with a small file, 10000 with a medium one, and 5000 with a large file), the average increase in time to delete or move a non-ELF file was 15.59% or $6.032\mu\text{s}$ when the file was cached. The time required to open a cached non-ELF file for writing similarly went up by 34.84% or $3.86\mu\text{s}$. For unsigned ELF files, the overhead of deleting or moving the file increased by 26.77% or $13.3\mu\text{s}$. The overhead of opening an unsigned ELF file for writing increased by 66.65% or $7.73\mu\text{s}$. All tests were performed on a Pentium 4 at 2.8GHz with 1G of RAM. While these percentage increases are high for opening a file, the amount of physical time required to open a file remains small. In the interest of retaining file-system compatibility with kernels not enabling bin-locking, we chose not to optimize the overhead of moving, deleting, and opening bin-locked files. By reserving one bit per file on the file-system for indicating whether a file is signed or not, this overhead could be brought down to essentially 0%.

When the file was not cached, the time required to perform an update, move, or open varies with the speed of the hard drive. Determining if a file is bin-locked requires three additional hard drive accesses. The first disk read is to bring in the data block pointers [157], the second is to read the ELF header, and the third is to read the section table. With a delete, the data block pointers need to be read from disk anyway, resulting in bin-locking requiring an overhead of just two additional disk reads. During prototype testing, the average increase in time required to both delete and move a non-cached and unsigned ELF file was 28ms. This overhead can also be brought down by tracking whether files are bin-locked on the file-system.

The cost of replacing a bin-locked file (i.e., the cost of validating the signature on a signed binary) is $O(n)$ in the proposed system (where n is the size of the file), an increase from $O(1)$ in a system not enforcing the protection mechanism. This overhead translates to 111.8ms on the test system for a 1M binary (with disk caching disabled[3] and using the older digital signature verification implementation as mentioned in Section 5.6.2). The overhead is apparently unavoidable, since the entire file must be hashed to verify a digital signature. We emphasize that this cost is only occurred during the install or upgrade of a bin-locked binary, not while performing normal tasks (i.e., executing an application).

5.6.5 Protection Against Current Rootkits

To verify that the bin-locking system was able to defend against rootkit malware, we attempted to install several Linux rootkits.[4] Linux rootkits can be grouped into two categories. The first category is those that use some method to gain access to kernel memory, installing themselves in the running kernel. These rootkits then operate at kernel level, hiding their actions from even root processes. The second category consists of rootkits that replace core system binaries. These binaries are often used by the root user in examining a system. Both classes of rootkits work to hide nefarious activities and processes on a compromised system.

We selected six representative Linux rootkits, two that modify the kernel and four that replace system binaries. Both kernel-based rootkits (`suckit2` and `mood-nt`) failed to install because of disabled write access to `/dev/kmem`. The `mood-nt` kernel based rootkit which we tested also attempted and failed to replace `/bin/init` (in order to re-initialize itself on system boot); this replace-

[3]Disabling disk caching involves writing other information to memory, causing the cached file to expire and be removed from disk cache.

[4]All Linux rootkits tested were from `http://packetstormsecurity.org/UNIX/penetration/rootkits/`

ment was denied by the modified kernel. The four binary replacement rootkits (`ARK 1.0.1`, `cb-r00tkit`, `dica`, and Linux Rootkit 5) were all denied when attempting to replace core system programs (e.g. `ls`, `netstat`, `top`, and `ps`). The bin-locking prototype provided protection against the modification of both application and system binaries. The fact that no rootkit was able to install is supporting evidence of expected protection functionality of the bin-locking system.

5.6.6 Reboots

Because the prototype requires a reboot to delete or move bin-locked binary files, the process of rebooting into a kernel which does not enforce bin-locking must be as usable as possible. We used the standard GRUB [127] boot loader to provide an option to the user as to whether or not to use the bin-locking enabled kernel; the user must then select one of the non-enforcing kernels from the menu during boot. Once booted into an alternate kernel, the user may delete and move bin-locked files, including those installed by malware (as discussed in Section 5.4.1). An open problem is how to persuade users to choose to use the kernel which enforces bin-locking. Creating a trusted interface between the kernel and user (e.g., see Ye et al. [193]) – one that cannot be subverted by malware – may help eliminate the reboot requirement; as discussed in Section 5.2.4, a hardware key is one such option.

5.6.7 Protection Against Downgrade Attacks

The downgrade attack involves an attacker replacing a recent version of a file with an earlier version known to contain vulnerabilities. To protect against this attack, we suggest deploying versioning (as discussed in Section 5.5.1). Should the particular implementation of bin-locking not support the versioning extension, developers can protect against the downgrade attack by introducing new public verification keys and expiring older keys when releasing a new version of the binary known to fix vulnerabilities. Since the key will have changed, none of the public verification keys in the newer binary (which is currently installed) would be able to be used in validating a public verification key in the old binary, and downgrade would be prevented.

Scheme	Proactive	Upgrades	File Types	Granularity
Bin-Locking (this chapter)	Yes	Yes	Program Binaries	File
Google Android v2.0 [5]	Yes	Yes	All	Application Package
Rootkit-Resistant Disks [30]	Yes	No	All	Hardware Token
Tripwire [86, 87]	No	No	All	File
Read-Only Media [92]	Yes	No	All	System

Table 5.1. Comparison of related file-system protection mechanisms. The granularity indicates to what extent the principle of least privilege is applied when modifying files.

5.7 Related Work

We first compare bin-locking with several closely related proposals and then discuss other related work in Section 5.7.4. Table 5.1 focuses on Google Android v2.0 [5], rootkit-resistant disks [30], Tripwire [86, 87], and using read-only media [92], comparing the approaches on whether they are proactive (prevent versus detect modifications), accommodate upgrades (without extra end-user effort), the types of files they protect, and the granularity of protection. While only bin-locking and Android accommodate program upgrades, the table is not the full story. We now discuss differences in more detail.

5.7.1 Android

In parallel to and independent of our work (but subsequent to publication of our preliminary design [189]), Google introduced a signing approach [61] in the Android platform [5] which closely parallels bin-locking. An application developed for Android v2.0 is packaged and signed with a private key created by the developer. As with bin-locking, there is no requirement for a public key infrastructure. Application updates under Android are allowed if *all* public verification keys in the new version are also in the installed version of the package, and can be used to verify the corresponding signatures in the new version (variation 2 in Section 5.5.3). In contrast to bin-locking, Android precludes new public verification keys being introduced during upgrade. While bin-locking signs individual binaries, Android signs application packages. Each application is copied into its own separate directory during install by the Android OS. The OS keeps track of application signatures and prevents applications from overwriting files outside their install directory. The Android approach protects all types of files, not just application binaries. Backwards compatibility requirements preclude bin-locking from assuming that all data associated with an application is installed into the same directory (e.g., configuration files are commonly all stored in `/etc` and binaries stored in `/usr/bin` on Linux [144]). The Android signing approach is a customized solution for the platform because of the constraints it puts on how and where applications are installed. Bin-locking comes as close as possible (in our view) to a general solution while preserving backward compatibility with current file-system layouts.

In Android version 1.6, each public key is tied directly to a signature it can be used to verify (see Section 5.5.3). This enforces that keys can not be added in subsequent versions of the application package (again, discussed in Section 5.5.3). The tying of keys directly to signatures instead of using a key prefix is an artifact of using the Java archive signer. Another artifact is that an application can be repeatedly signed with previous signatures remaining valid (recall, bin-locking does not allow this because it also signs all bin-locking metadata and keys). In preventing signatures corresponding to new private keys from being added to subsequent versions of the application, an attacker is prevented from signing a valid package to produce a new valid package including a signature corresponding to a private key that the attacker possesses.

5.7.2 Rootkit-Resistant Disks

Rootkit-resistant disks by Butler et al. [30] relies on the user inserting a hardware token every time an area of the disk "protected by" that token is updated. New changes written to disk with the token inserted are marked as requiring the presence of that token in order to be modified. While rootkit-resistant disks protect a much larger range of file types than bin-locking, a knowledgeable user is required to insert the hardware token whenever any write operation is performed to the protected files (including updates). To protect every application separately, a different hardware token would need to be used for every application installed on the system. If only one token is used, then any application can modify any other application arbitrarily as long as the hardware token is inserted. A single-token system fails entirely if the user is ever tricked into running malware during the time the token inserted (including if the token is inserted after malware has started running).

Combining rootkit-resistant disks with bin-locking would eliminate the most common instance which would require the token – a software update. A combined solution would also increase the granularity of the rootkit-resistant disk solution while protecting all types of configuration files – a protection the current bin-locking approach does not provide.

The general approach of restricting writes to the file system as a method to combat malware is longstanding [50]. Simply asking the user for authorization every time a file is modified on the system results in an unworkable solution, even for experienced developers. Rootkit-resistant disks go a long way toward creating a workable solution. Bin-locking takes the approach further in allowing updates to be performed without user intervention.

5.7.3 Tripwire and Read-Only Media

Tripwire [86, 87] records cryptographic checksums for all files on a system to detect what files are changed by malware (by comparing against the current checksum). Read-only media [92] prevents any change from being made to the drive while the system is running, allowing the user to revert to a known-good state by simply rebooting the system. While Tripwire and read-only media differ from each other in their ability to prevent changes to the file-system, they share many characteristics, the most prominent being the way they deal with software installs and upgrades. With read-only media, the install or upgrade must be made on a system with writable media and then a new version of the read-only media is created – a potentially time-consuming process. With Tripwire, all changes to the file-system are flagged as potentially bad and the user must verify that each file modification is indeed acceptable (also a time consuming process). In both cases, security patches become troublesome to install. With Tripwire, the user has the option of verifying that an application does not overwrite core system binaries during install or upgrade – the same is not the case when updating read-only media. Tripwire does not prevent the modification of a file; it only detects these modifications.

5.7.4 Other Related Work

Related to work by Butler et al. [30], SVFS [195] also protects files on disk at the cost of running everything in a virtual machine. Software updates and installs are not addressed by SVFS. Strunk et al. [161] proposed logging all file modifications for a period of time to assist in the recovery after malware infection. Their approach does not prevent binaries from being modified in the first place. By combining their approach with bin-locking, logging can be restricted to configuration file changes – decreasing disk space requirements.

There have been many attempts at detecting modifications to binaries (in addition to Tripwire, discussed above). Windows file protection (WFP) [113, 37] maintains a database of specific files which are protected, along with signatures of them. The list of files protected by WFP is specified by Microsoft and focuses on core system files. WFP is designed to protect against a non-malicious end-user, preventing only accidental system modification. Pennington et al. [130] proposed implementing an intrusion detection system in the storage device to detect suspicious modifications. All these attempts rely on detecting modifications after the fact. While WFP is capable of handling updates, the other solutions do not appear to directly support binary updates.

Apvrille et al. [8] presented *DigSig*, an approach which also uses signed binaries in protecting the system. They modified the Linux kernel to prevent binaries with invalid signatures from being run (as opposed to the bin-locking approach of preventing the modification). Under DigSig, all binaries installed must be signed with the same key. While the use of a single key may work for corporate environments deploying DigSig, it does not seem well suited to decentralized environments. DigSig also relies on a knowledgeable user to verify all updates to binary files (similar to Tripwire) before signing them with the central key.

The approach to bin-locking differs from that of van Doorn et al. [174] (and indeed many other signed-executable systems such as that by Pozzo et al. [134] and Davida et al. [44]). In these systems, the installation (or running) of binaries is restricted by whether or not the application is signed with a trusted key. In contrast, bin-locking does not restrict the introduction of new executables (those with new file names) onto the system and does not rely on any specific root signature key being used, or external notions of trusted keys; it does not rely on any centralized PKI.

With all the approaches described (with the exception of that by Butler et al. [30] when using multiple tokens), it seems one common pitfall is that any application performing an update or install will have permissions sufficient to modify any other binary on the system. Some proposed systems attempt to mitigate this threat by assuming a vigilant and knowledgeable user will verify all changes to binaries. They rely on this user to never be tricked into installing a Trojan application. We believe that by differentiating between files originating from different developers or organizations, bin-locking can rely less on vigilant and knowledgeable users to protect some parts of the system. All approaches except bin-locking treat upgrades the same as new application installs.

While policy systems such as SELinux [102, 76] have the capability to restrict configuration abilities, the overhead of correctly configuring a policy for every application (including every installer) makes this approach unrealistic in many environments. Bin-locking allows binaries to be protected based on who *developed* (or *created*) them, a property not easily translated into frameworks such as SELinux. While projects such as DTE-enhanced UNIX [180] and XENIX [167] restrict the privileges of root (reducing the risk of system binaries being overwritten), installers (and even upgrades) are still given full access to all binaries on disk.

The OpenBSD `schg` [93] and ext2 immutable [169] flags are similar to bin-locking in that they prevent files from being changed, moved, or deleted. These flags, however, do not allow an application binary to be updated, resulting in a system more akin to read-only media (see Section 5.7.3).

Conventional code-signing involves verifying the author (code source) before software is run [142]. Code-signing approaches generally do not restrict what the software can do while running. When applied to installers, code-signing allows a user to verify the source of the software they are about to install (and that the software has not been modified since the vendor signed it) – the same is true for package managers [31]. In both cases, the signature applies to the entire package (not to the individual binaries) and does not end up restricting which binaries either the installer application or installed program can modify. While some systems may maintain a cryptographic hash for files installed, these hashes are more akin to those used by Tripwire (see Section 5.7.3). Hashes alone are insufficient for tying two versions of a binary to the same source. While the bin-locking approach can prevent binaries modified during distribution from being installed as an upgrade, we do not focus specifically on this problem as do Bellissimo et al. [22].

5.8 Final Remarks

The approach discussed in this chapter provides a convenient method for tying the product of a developer's efforts to the developer. While we took the approach of tying binary files the developer created to a key the developer holds, the approach is equally capable of tying other digital objects (or indeed, collections of objects such as an application package) to the developer key. Should the specific implementation created by the guardian not support unsigned binaries, all developers would be forced to sign their applications. We believe such a restriction would not overburden developers, since the process of a developer signing an application does not require any third party. Android already enforces such a requirement, where all applications must be signed before being installed. The approach of being able to isolate the work of a developer from being modified by others without the use of a centrally managed PKI provides a mechanism for restricting the abilities of malware. The approach does not rely on the end-user for enforcement, protects all applications using the approach, and is implemented by a guardian (the OS developer). It therefore follows the thesis goal of providing a guardian enforced mandatory access control mechanism which can be deployed.

6 `Configd`: Reducing Root File-System Privilege

While the principle of least privilege dictates that the privileges assigned to a process should be no more than the privileges required to perform the designed task, the standard exercising of root privilege in order to install applications does not follow the principle. While some progress has been made by encouraging users and daemons not to run as root, the same cannot be said for installers – perhaps the most common use of root privilege in the current computing environment is for system reconfiguration (i.e., installing, uninstalling, or upgrading software). In this chapter, we pursue reducing the file-system privileges of root in order to better protect a system against abuse.

The actions performed by any user (including root) on a system can be partitioned into two classes. The first involves actions related to performing day-to-day operations on the system (e.g., writing a paper, browsing, reading email, or playing a game). Such actions typically do not have a lasting impact on the state of the system (modulo data file creation and deletion). The second class involves actions related to changing system configuration. We define the *configuration state* of a system as the set of programs installed, as well as the global configuration related to each program. In order to survive reboot, both the programs installed and all global configuration state must be saved into the file-system, and hence we focus on those configuration operations having a direct visible effect on disk.

In this chapter, we focus on preventing one application from modifying another's file-system objects on disk. We focus exclusively on system-wide application data, configuration, and binary files (i.e., we do not consider user data files in this chapter). The common protection long used in practice is to limit write access to application file-system objects (e.g., files including binaries, directories, symbolic links, and other objects that are part of the file-system) to root [42]. This protection mechanism fails to prevent abuse by applications during install, upgrade, or uninstall. In today's computing environments, it is only realistic to treat any two applications on a system as mutually untrustworthy. Given this updated threat model, we further subdivide configuration in order to *encapsulate* applications – by this we mean that while it may be possible for one application to read the binary, data, and configuration files belonging to another application, it is not possible to modify another's files on disk. In contrast, current desktop approaches for software installation do not prevent an application from modifying or deleting file-system objects related to or created by an unrelated application. Our restriction and division of root file-system permissions addresses this problem, without requiring any radical change in file-system layout (e.g., applications can still install their binaries in a common location such as `/bin`). As a direct result, applications are better protected from malware and other applications, even those running with root privileges. The approach discussed in this chapter is distinct from that discussed in Chapter 5, in that we focus on enforcing additional protection mechanisms on all types of configuration related files, not just application binaries.

In our design, the preliminary ideas of which were outlined in a workshop paper [191], the ability to modify arbitrary objects (beyond simply files) on the file-system is removed from root and reassigned to a process running with a new *configuration privilege*. This process in turn can be used to prevent one application from modifying the file-system objects related to another. In creating a distinct configuration permission, the configuration tasks currently performed under root privilege are separated from the everyday tasks. Daemons, applications, or installers running as root no longer automatically inherit configuration privilege. `Configd` acts as a reference monitor for system configuration.

Our implemented prototype system,[1] using Debian 5.0 as the base environment, consists of a modified Linux kernel which restricts updates to designated file-system objects, a modified Debian package manager, and a user-space daemon (called `configd`) which is responsible for protecting an application's file-system objects from being modified by other applications. A control point made available in `configd` allows each configuration related file-system modification request to be examined, and either authorized or denied. As we explain in detail later, the prototype successfully prevented installation of current rootkit malware while having an imperceptible overhead to the end-user. While our discussion and prototype focus on Linux, we believe the approach can be adapted to Windows, Mac OS X, BSD, and other operating systems. Indeed, `configd` implemented on Windows could also protect the Windows registry (since it is stored on disk).

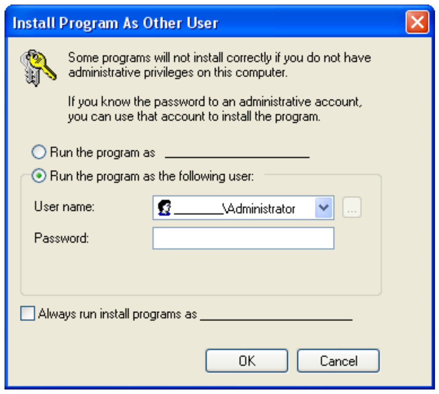
[1]This prototype is distinct from the prototype in Chapter 5.

6.1 Background on Installers

Our approach of limiting abuse of root privilege as it relates to configuration changes of designated file-system objects is most relevant to the case of software installs, upgrades, and uninstalls. For context and to highlight the problem, we review existing approaches to installing software on a desktop.

6.1.1 Application Installers

The most common approach to installing applications on commodity desktops and servers is through the use of an application installer. The installer is a binary or script, often written by the same company or individual that developed the application. Its purpose, when run, is to place the various file-system objects associated with the application in the correct location and configure any system parameters (e.g., in some cases to ensure the program gets run during boot). Application installers are typically given complete control over the system during install, with users encouraged to run them with full permissions as shown in Figure 6.1. Whenever an application installer is run on a typical system, the entire system is opened up for modification by the installer. If the installer is malicious, or can be compromised [22], the entire system can become easily compromised. This approach does not prevent one application's installer from modifying the file-system objects of another application.



Install Program As Other User

Some programs will not install correctly if you do not have administrative privileges on this computer.

If you know the password to an administrative account, you can use that account to install the program.

Run the program as ______

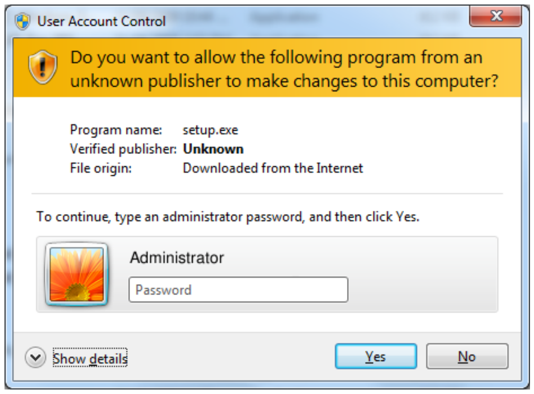
[✓ Run the program as the following user:]

User name: ______\Administrator

Password:

Always run install programs as ______

(a) A Windows XP prompt



User Account Control

Do you want to allow the following program from an unknown publisher to make changes to this computer?

Program name:	setup.exe
Verified publisher:	**Unknown**
File origin:	Downloaded from the Internet

To continue, type an administrator password, and then click Yes.

Administrator

Password

(b) A Windows 7 prompt

Figure 6.1. Windows prompts to run an installer with administrator privileges.

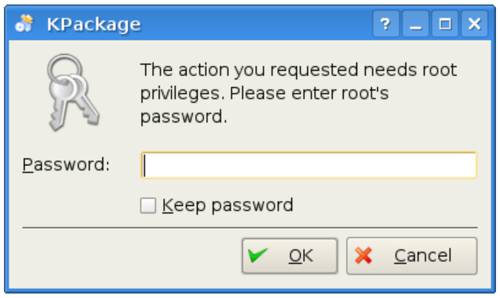
Application install scripts, like those executed through the `make install` command on many open source projects, are a slight variation of the application installer. When told to install to a user's home directory, they do not require administrator privileges. They still require administrator privileges, however, when attempting to install to a location controlled by the administrator. Using `make install` does not prevent modification of file-system objects belonging to other applications installed by the same user. While Windows encourages following the principle of each application installing into its own directory on the file-system, the practice of running application installers as root leaves the principle unenforced.

6.1.2 Application Packages

Package managers are typically provided by an operating system (OS) or OS developer to ease development of an application installer for the platform. Instead of writing an application installer from scratch, the application developer creates a package using the package manager APIs and following the rules set by those who developed the package manager. Typically, the package consists of data used by the package manager, as well as files to be installed and scripts to run as part of the install. Common install operations are taken care of by the package manager. While the development of package managers has resulted in installers transitioning from being executables to packages (e.g., various Linux packages [31], Microsoft Installer packages [119], and Apple application packages [162]), most of these package managers still allow for executing arbitrary binary code or scripts contained in the package being installed, resulting in the same level of risk to the system as if an executable was run to perform the install. If root permission is requested by the package (or required by the package manager, as is the case with Linux packages), and the end-user enters an appropriate password when prompted (as users are now well trained to do so upon request), these scripts are run as root. The use of application packages does not prevent an application from modifying arbitrary file-system objects. One example of a malicious Debian package was a screensaver posted on `http://gnome-look.org`. The package, when installed, would also download and install a bot onto the local machine [170, 176].

6.1.3 Apple Bundles and Packages

With Mac OS X, Apple introduced a new method for installing commodity applications other than application packages: application bundles. This results in two approaches on Mac OS X for installing software.


KPackage

The action you requested needs root privileges. Please enter root's password.

Password:

Keep password

Figure 6.2. A KPackage prompt to obtain root privileges before installing a package.

Application Bundles

Application bundles are similar to packages as discussed above with two key differences. The first is that all file-system objects in the bundle will be installed into the same sub-directory by the OS (akin to the approach used on Android described in Section 6.2.2). The second is that scripts are not run by the OS during install. Typically, application bundles are installed through a drag-and-drop style copy operation. Commonly distributed as disk images, application bundles greatly increase the security of the system against malicious application installers. Unfortunately, Apple still supports a second approach for installing applications and relies on the software developer to choose between them. The malicious developer is unlikely to distribute his software as a bundle when distributing a package (see below) is possible. Legitimate applications are distributed as both bundles (e.g., Mozilla Firefox) and packages (e.g., Adobe Flash).

While application bundles remove the ability for many applications to obtain root privileges, they do not mitigate the entire threat. Any application which obtains root privileges (maliciously or otherwise), even if installed as a bundle, can still modify any file on the system.

Application Packages in Mac OS X

For more complex applications, additional actions (other than simply copying the files into the application directory) may need to be performed by the installer during the install process. For these applications, Apple provides an application package framework as discussed in Section 6.1.2.

The status quo across multiple operating systems is to “restrict” installers by requiring the user authorize an installation by entering an administrator password before the installer can run. Once the installer is given “blind” full permission to the system, the user must trust that the installer does not abuse its privilege (since it is difficult to tell what the installer actually does with its root privilege – thus we call it blind trust).

6.2 Approaches for Encapsulating Applications

In this section, we discuss several approaches for encapsulating applications, using the definition of encapsulation from page 6. While we choose to implement the GoboLinux approach in `configd` (as discussed later in this chapter), the alternative approaches discussed in this section are equally viable. We summarize the three approaches in Table 6.1. Note that although Table 6.1 indicates that for GoboLinux, application upgrades are not supported, the use of the GoboLinux approach in `configd` is done in such a way that application upgrades are supported.

Attribute	GoboLinux	Android	Apple App Store
Application Upgrades	Not Supported	Supported	Supported
Allows Install Scripts	Yes	No	No
Application Deployment	Unrestricted	Unrestricted	Restricted

Table 6.1. Characteristics of current systems which encapsulate applications on disk.

6.2.1 GoboLinux

In GoboLinux (version 014.01) [121, 122], each application is installed into its own directory (similar to Android). The actual install of a program is done through calling three scripts. `PrepareProgram` is responsible for creating the base directory where the program will be installed. `CompileProgram` takes a compressed archive of the program source, configures it with the appropriate flags so it will be installed into the directory prepared for it by `PrepareProgram`, compiles the program, and installs it. `SymlinkProgram` creates symbolic links to the various program binaries, libraries, and settings. For backwards compatibility, the common Unix directories (e.g., `/usr/bin`, `/sbin`, and `/lib`) are also linked to `/System/Links`, which is in turn populated with symbolic links for each of the applications that have been installed. To decrease the ability of an application to escape its assigned application directory, the `make install` command is run under a special user ID [120]. This user is only allowed to modify files under two directories: the one in which the application is being installed to, and the one it was compiled from [121, 122]. In restricting an application during both compile and install, the other applications on the system remain protected against modification.

For those applications which are not complete programs in themselves but instead extensions to other applications which are already on the system (e.g., the PHP module is often installed as an extension to the web server Apache), the base application needs to be made aware of the extension. In GoboLinux, the configuration of each application is stored in the shared tree `/System/Settings/`. All files in this directory are symbolic links that point back to the settings folder, which is a sub-directory of where the application was installed. The base application provides a directory under `/System/Settings` where a module can register itself (through `SymlinkProgram`). Many distributions other than GoboLinux have also adopted this as the method of installing extensions into a base application (although the exact path to the configuration will change). Another related example involves services which should be started on system boot. On Debian based distributions, the accepted location for a script responsible for starting a service is in the directory `/etc/init.d`. GoboLinux places scripts responsible for starting the various system services in `/System/Links/Tasks`.

GoboLinux depends heavily on symbolic links being placed into the above mentioned standard directories for extensions to applications. The base GoboLinux executable `SymlinkProgram` is responsible for updating this directory tree based on the layout of files in any particular application directory (GoboLinux does not allow application installers to directly modify the symbolic links in shared directories).

A limitation of GoboLinux is that it does not cleanly support upgrades (or security patches) to an application. Each upgrade is treated as an install of a new version of the application, resulting in each version being installed into its own directory on disk. The job of sorting out which version of an application should be used by default on a system is left to `SymlinkProgram` (the PATH environment variable is set to point to `/System/Links/Executables`, a directory maintained by `SymlinkProgram`). The sharing of configuration files between different versions of an application is left up to the individual application. Indeed, each version of an application has its own copy of the configuration files stored in a `Settings` sub-directory of the application directory, alongside the various sub-directories for each version of the application which is installed.

While we choose in this thesis to maintain the current file-system hierarchy, negating the need for symbolic links, changing the file-system hierarchy and incorporating the functionality of SymlinkProgram into configd is another approach which can be used to encapsulate applications. The approach used by CompileProgram, where associated installation scripts are run with permission to write only to files directly associated with the same application, restricts installation scripts while still allowing them to exist. We discuss an implementation of this approach in configd in Section 6.4.4.

6.2.2 Android

On the Android (version 2.0) platform [5, 61], each application package is assigned its own directory and unique user id. While Android uses the Linux kernel as its base, being a single user platform, Android remapped the traditional user accounts to restrict communication between applications. The Android application installer ensures that each application is restricted to making file-system modifications in the directory it was installed into. Unlike the file-system hierarchy standard [144] as used on Linux, there are no shared directories for storing binaries, libraries, configuration files, and other elements. Android benefits greatly from the ability to mandate a file-system layout which restricts each application to single sub-directory on the file-system.

The Android (version 2.0) platform only allows a new version of an application to be installed over top of the old if all public keys in the new version are also contained in the old version already installed (new keys cannot currently be introduced during an upgrade). During the install of an application, the application itself is not given a chance to run any installer scripts as administrator – greatly restricting the damage a particular application can do to files belonging to other applications. The platform does a good job of preventing one application from modifying another's files.

With additional work, the Android approach can be adapted to the standard Linux file-system hierarchy [144]. Instead of storing all files related to an application in a single directory, a database could be maintained which maps each individual file to the application it is associated with (the Debian package manager already keeps such information), as well as a list of public keys which are used to verify the next version of the package. The Android approach does not support scripts being run as part of the installation process (similar to Apple bundles not supporting scripts; see Section 6.1.3). Combining the Android approach with GoboLinux, however, allows the execution of installation scripts which can modify the configuration of the application being installed while still preventing other applications from being modified.

6.2.3 Apple Application Bundles

While the general application bundle is discussed in Section 6.1.3, a number of restrictions were made by Apple for bundled applications targeting iPhones (up to those released in January 2010) [7]. The biggest change is that each application installed onto the iPhone is limited to making file-system modifications only in the directory it was installed into. This includes limiting an application to reading and writing data files to an application specific area of the file-system (similar to Android).

In contrast to Android, Apple application bundles targeted to the iPhone are not signed with the key of the software developer. Instead, each application must be signed by Apple before it can be run on an iPhone (we ignore "jail-broken" iPhones in our discussion). Before signing an iPhone application bundle, Apple examines the application to ensure it meets their criteria [6]. For application bundles designed for Mac OS X, Apple has no such restriction that the bundle be verified and signed by Apple before it can be installed.

6.3 The Protection Mechanism

Our main objective in this chapter is to divide root privilege so that programs, such as installers, cannot take advantage of overly coarse access controls to abuse the privileges they have been given. The design of our approach is subject to several self-imposed constraints. We believe that, to be viable, any alternate approach for restricting file-system privileges on the desktop at a per-application level would need to fulfil the following considerations.

1. **Compatibility with current file-system layouts**. In designing a Linux-based prototype of the proposal, our goal was to avoid requiring redesign of the current file-system hierarchy [144] in favour of a solution compatible with the current file-system layout. Applications are encapsulated, being protected against modification while retaining the current file-system layout – installing files to directories shared with other applications. In contrast, GoboLinux [122] and Apple (in Mac OS X) did redesign the file-system hierarchy. Their motivations were apparently to impose cleanliness in restricting each application to its own directory. While the separation of each application into its own directory may simplify the challenge of restricting configuration changes on a per-application basis, a backwards compatibility layer is still required to support applications not designed for the new layout.

2. **Minimal impact on day-to-day operations**. Most of the time, a computer is used to perform day-to-day tasks (run applications) with a constant configuration of the applications and operating system. Occasionally, its configuration is modified in order to expand/modify the tasks it can perform (e.g., applications are installed, updated, removed, or reconfigured). Our proposal (and indeed any alternate `configd` approach) should impose no noticeable impact on such day-to-day operations, with no changes to regular user work flow.

3. **Backwards compatibility for current installers**. We introduce new restrictions on an application's ability to modify file-system objects. These restrictions will typically influence the install, upgrade, and removal of applications. It is unrealistic to assume that all installers will be modified in parallel during deployment of such a solution. Backwards compatibility is therefore critical for incremental deployability.

   In the prototype, to ensure that all file-system operations modifying configuration related files are handled by `configd`, a kernel security module was written which redirects all requests for modification to configuration related file-system objects to `configd` for verification. Any application not written explicitly to communicate with `configd`, including all scripts run as part of an application install, would have their file-system operations rerouted. In addition to implementing a backward compatibility layer, the prototype uses a modified package installer for Debian – compatibility was maintained with standard Debian packages. We foresee any alternate use of the control point provided by `configd` as also having to support current install methods (subject to the constraint that the installer does not attempt to break the per-application encapsulation restrictions).

4. **Usability**. Our focus in this thesis is on providing a solution which can be used by non-expert users, and to avoid forcing upon users choices which they are ill-equipped to respond to correctly. Our solution achieves this goal, allowing an applications' file-system objects to be protected against modification during install, upgrade, uninstall, and at run-time, without presenting the user with complex choices. While our prototype solution did leave enabled the option of querying the user about file-system operations, this feature can be safely disabled (as discussed in Section 6.4.5).

5. **Other considerations**. We assume that the user of a computer system can be trusted to not be malicious. The proposal, therefore, does not protect against physical attacks, such as rebooting into a kernel that does

not enforce the proposed protection mechanism.[2] Its security also assumes that applications cannot obtain kernel level control of the system,[3] although on most current systems, any application running with root access can modify the running kernel. This assumption therefore relies on the mechanism presented in Chapter 4.2 to be in place.

6.3.1 A Division of Root Privileges

To build a system designed to reduce root abuse of file-system privileges, we first separate configuration related activities (those configuration actions affecting the applications installed or their global configuration as stored on disk). We then further subdivide the configuration privilege to remove the ability of an application installer to modify any file-system object other than those which are part of the application being installed (or upgraded/removed). While Chapter 5 focused on enforcing additional protection mechanisms on binary files using bin-locking, we now focus on enforcing additional protection mechanisms on all types of configuration related files (through the use of `configd`), not just application binaries.

Our prototype design consists of two main elements: a kernel extension and a user-space daemon. The user-space daemon is responsible for the bulk of the work, namely, ensuring that one application cannot modify files related to a different application. The kernel is responsible for denying (or forwarding) requests to modify protected *file-system objects*, by which we mean files (including binaries), directories, symbolic links, and other objects that are part of the file-system. Protected file-system objects (which we call *c-locked* file-system objects, short for *configuration locked*) are designated (marked) as such by the user-space daemon (an alternate method of protecting file-system objects is by using a union file-system to redirect writes [188]). Any application file-system object marked as part of the system configuration (and hence to be protected against modification by other applications) must be so designated. While we leave open the exact set of c-locked file-system objects, we view the set to include shared libraries, executables, system configuration files, start-up scripts, and other file-system objects which do not change as a result of day-

[2]While some have proposed that the user cannot be trusted [177, 101], our work avoids declaring the user as the enemy and preventing them from modifying their own system. We favour persuasive tactics as a tool to encourage users to properly maintain their system while not taking the control out of their hands.

[3]We define the kernel (and hence kernel level control) to include only those aspects running with elevated CPU privileges (ring 0 privileges on x86); this definition does not include core system libraries installed alongside the OS but run in user-space.

to-day system use. In effect, the system configuration on disk includes the set of applications installed as well as each application's files which are required at run time. Only a process holding a newly introduced *configuration permission* is allowed to modify the system configuration on disk (and hence c-locked file-system objects).

The exact distribution of duties within our design has the kernel responsible for:

1. Restricting to programs running with configuration permission the ability to delete, move, and write to c-locked file-system objects. By design, "root" is not allowed to make arbitrary changes to c-locked file-system objects (including the kernel image) – changes are limited to processes running with configuration privilege.

2. Restricting the ability to obtain configuration permission. In our prototype, this is done by allowing only a single process, the configuration daemon (`configd`) to have configuration permission.

3. Restricting the ability to control the process running with configuration permission (e.g., by not allowing `configd` to be killed or modified by a debugger).

In restricting configuration permission to a single daemon, we introduce a chokepoint within which we can further subdivide file-system access by application.[4] We perform the actual subdivision in the above-mentioned configuration daemon `configd`. It performs the following operations:

1. Respond to requests for configuration changes from processes running on the system. In our prototype, requests were at the granularity of package operations, but our design could be easily modified to handle other granularities.

2. Designate file-system objects as c-locked. It marks a setting in a data structure associated with the object to denote this designation. In our prototype, every file installed when upgrading or installing a package is marked as c-locked.

3. Perform authorized changes to the configuration of the system.

[4]While it is possible to use the kernel as the chokepoint, our preliminary exploration in this direction suggested that implementing the functionality required to further subdivide root on a per-application basis directly in the kernel introduces functionality into the kernel which is already available in user-space.

We now discuss in more detail how the two elements work together to restrict the ability for an application to modify file-system objects belonging to another application.

6.3.2 Linux Kernel Protection of C-Locked File-System Objects

How a kernel handles file-system objects can directly affect the security of c-locking. In the Linux kernel, the key file-system data structures directly related to the protection of file-system objects by `configd` are the *directory entry* (or dentry) and inode [157]. The inode data structure contains most of the information related to the file data and meta data (e.g., the traditional Unix file-system permissions read, write, and execute) and pointers to the actual blocks on disk which hold the file contents. The dentry contains information related to the specific instance of a file in a particular directory, including the name of the file (as it appears in that directory) and a pointer to the inode. For the purposes of c-locking, the dentry inherits the c-locked status of the inode. If the inode is marked as c-locked, then the directory entry can be deleted or moved only by `configd`. File operations on an inode which is not c-locked are restricted through current access-control restrictions (including traditional Unix file permissions). Figure 6.3 demonstrates the relationship between inodes and dentries.

Symbolic Links. Symbolic links are directory entries in Linux pointing to an inode containing a path string. When opening a symbolic link, the kernel retrieves the path name from the symbolic link inode. It then follows the retrieved path name to obtain another dentry and inode (which is either yet another symbolic link or some other element such as a file or directory). The proposed system supports either c-locking the symbolic link, the object it points to, or both.

Hard Links. A hard link is a directory entry in Linux pointing to the same inode as another directory entry. As with regular files, because the inode itself contains the c-lock flag, any hard link pointing to the inode inherits the c-locked attribute associated with the inode. An attacker does not gain modification privilege by creating a hard link to a file-system object protected by c-locking. The ability to create a hard link to a c-locked file is restricted, being either allowed or denied by `configd`.

Directories. A directory is an inode which instead of pointing to file data, points to a list of dentries. While previous approaches [30, 195] focused on protecting files more than directories, there are cases in which a directory should

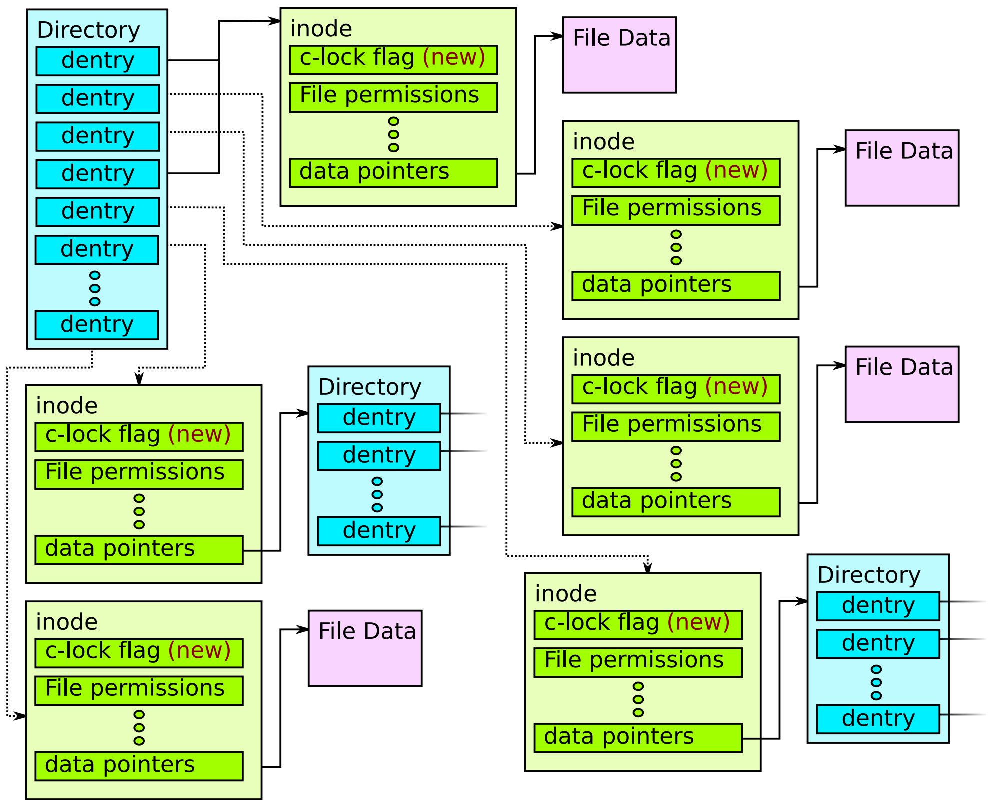


Figure 6.3. File-system data-structure layout including new c-lock flag.

be protected. As an example, during start-up the Debian `/etc/rcS.d` directory is accessed and every file (or file pointed to by a symbolic link) in this directory is run. Any malware installed into this directory would be started automatically during system boot. The proposed system can protect directories since they can be c-locked in the same manner as files and symbolic links.

6.3.3 Configd

The prototype `configd` is designed to subdivide root file-system permissions on a per-application basis. In our framework, `configd`, or its equivalent, becomes a chokepoint which applications must use in order to modify c-locked file-system objects (and hence the configuration of the system). This is illustrated in Figure 6.4. To enforce that `configd` is the only way that c-locked file-system objects can be modified, the kernel grants the new configuration permission to `configd` alone. By delegating this privilege to `configd`, the kernel need not know about every application on the system or what file belongs to which application; it need only recognize that a specific file is c-locked and leave the handling of this file to `configd`. The rules enforced by `configd`, in turn, are designed to be set by a guardian, during development of `configd`.

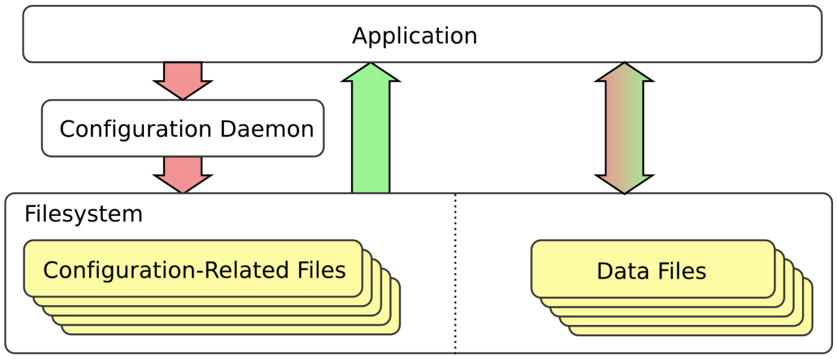


Figure 6.4. An illustration of `configd` as a chokepoint. Applications can continue to read and write to user files, but can only read from configuration related files.

`configd` must be started early during the boot process (`configd` itself restricts changes to the boot process). Once `configd` has started, other programs are prevented by the OS kernel from obtaining configuration permission. The design of `configd` takes advantage of the temporal nature of software installs. At the time of installation of application software, it is assumed that the operating system and `configd` are already installed and running. This assumption is reasonable if `configd` is made part of the OS or core system.

Example `Configd` Rule Set

While the core `configd` approach can use any number of different protection mechanisms for separating files on a per-application basis, we chose to depend on Debian packages (and indeed, the package manager – `dpkg`) in our solution. Because we used Debian as the base system in which to implement c-locking, the safe operations performed by the prototype are customized to that environment (e.g., we use the same file extensions as `dpkg`).

In Debian, the sequence of operations performed by the Debian package manager (`dpkg`) when installing any new file (during the install or upgrade of a package) is:

1. Create a backup of file-system object *A* by hard linking *A.dpkg-tmp* to the same inode as *A*.

2. Extract the new file which will replace *A* into *A.dpkg-new*. If the new file-system object being installed is a hard link, instead create a hard link called *A.dpkg-new*.

3. Move *A.dpkg-new* to *A*.

4. Remove *A.dpkg-tmp*.

Under the assumption that the kernel enforces c-locking according to a flag set in the inode, the following operations related to modifying c-locked file-system objects are allowed by the prototype. We have determined that these rules do not allow one application to arbitrarily modify file-system objects associated with another application, and hence can be considered "safe."

1. If *A* is a c-locked file-system object that does not end in *.dpkg-tmp*, then creating a hard link *A.dpkg-tmp* that points to the same inode as that pointed to by *A* is allowed.

2. Any c-locked file-system object ending in *.dpkg-tmp* may be deleted. During prototyping, it was determined that no permanent file-system objects have names that end in *.dpkg-tmp* and hence this operation does not allow a permanent file associated with an application to be deleted.

3. If *A* is a c-locked file and *A.dpkg-tmp* is hard linked to the same inode as *B*, then creating a file *B.dpkg-new* that is hard linked to the same inode as *A* is allowed.

4. If *A* is a c-locked file-system object whose contents under a cryptographic hash have the same value as *A.dpkg-new*, then *A.dpkg-new* can replace *A* (i.e., if *A* and *A.dpkg-new* contain the same data). We do not mandate any specific cryptographic hash algorithm, other than to stipulate that, at minimum, it must have second pre-image resistance [109] (our `configd` prototype currently supports SHA-1 and can be easily expanded to support others).

5. If *A* is a c-locked file containing one or more public keys and *A.dpkg-new* is another file containing a digital signature verified by using a public key in *A*, then *A.dpkg-new* is allowed to replace *A*.

6. If c-locked file *A* is associated with package *PKG* and is not associated with any other package installed on the system, then when upgrading package *PKG*, *A* may be modified.

7. If c-locked file *A* is associated with package *PKG* and is not associated with any other package installed on the system, then install scripts associated with package *PKG* may modify *A*.

8. All other operations involving a c-locked file are considered to be potentially dangerous. They can either be presented to an expert user for additional oversight, or simply denied (as discussed in Section 6.4.5).

Of the “safe” rules, some merit further discussion. Rule 3 was created as a result of the way two files that are hard linked together are updated by the Debian package manager. For this rule to apply, *A* and *B* would need to have been linked together (and subsequently *A.dpkg-tmp* created through rule 1). They must have been part of the same package (rule 6) or (if enabled) the user must have allowed *A.dpkg-tmp* to be hard linked to *B* (rule 8). Rule 3 limits hard linking to files distributed in the same package.

Rule 5 adapts concepts used by the Android OS [5], but to the file (as in Chapter 5) as opposed to package level. The rule relies upon the use of public keys and signatures, but does not rely on a PKI.

Rule 6 allows the modification of all files associated with a package when a new version of the package is installed. The rule requires that `configd` keep track of packages installed on the system, as well as which files are associated with which packages. We discuss the semantics of how our prototype handled packages in Section 6.4.3. It is important to note, however, that a package cannot modify any file on the system simply by asserting ownership of the file (through including the file in the list of files the package is associated with). Any file that is listed as belonging to more than one package cannot be arbitrarily modified by any package. The option still exists, however, for a file associated with more than one package to be updated through rule 5.

Rule 7 restricts, through exclusion, the files that an install script can modify. The rule is borrowed from GoboLinux [122] and discussed in more detail in Section 6.2.1.

Our testing confirmed that the above rule set allows upgrade operations performed by `dpkg` to be automatically allowed. While the option of querying the user with remaining operations was left enabled in the prototype, we found in testing that during upgrades, the user is not queried at all (see Section 6.4.5).

6.3.4 Incremental Deployability

While the approach discussed in this chapter must be enforced during every package install, upgrade, and uninstall in order to protect other application’s file-system objects, the approach does not need to be installed on all systems simultaneously in order for benefits to be realized. Systems choosing to implement `configd` and enforce restrictions on c-locked files will immediately realize the benefits of application encapsulation.

6.4 A Prototype Implementation

6.4.1 Kernel Extensions

To support c-locked file-system objects, we used the extended attributes functionality [168] of file-systems such as ext3 and XFS. This is the same approach used by SELinux [155]. In so doing, the underlying file-system-specific data structures do not need to be modified. The extended attributes are tied to the inode. We used the *trusted* extended attribute name space because it supported setting extended attributes on symbolic links. We created our c-locking protection mechanism as a Linux Security Module [102, 187]. The kernel implementation was approximately 2200 lines of code, including the backward compatibility layer. A new device node was used as the interface between the user-space `configd` and the modified kernel, allowing communication between the two. The process of opening the device node initiated c-locking protection in the modified kernel. The kernel understands and responds to several commands sent by the user-space `configd` through the new device node including:

- **release**. Because our kernel allows only a single process to obtain configuration privileges at a time, this command was included to allow `configd` to be stopped, upgraded and started during testing. On a production system, this command can be safely disabled.
- **freeze**. Freeze all processes on the system except for `configd`, to prevent race conditions as `configd` performs file-system changes. It also prevents processes from interfering with the user interface in our prototype (see Section 6.4.5).
- **thaw**. Unfreeze all frozen processes on the system, allowing them to continue executing.
- **noraw** and **raw**. Disable/enable raw write access on the hard drive device denoted by the associated options *major* and *minor*. This was used to prevent applications bypassing `configd` by writing to the underlying hard drive sectors associated with c-locked file-system objects.

The kernel also sends several commands to the user-space daemon `configd`, including:

- **plugin**. A USB token containing a magic value associated with `configd` has been inserted. The kernel is responsible for disallowing write access to the partition containing the magic value – on machines with `configd` enabled, only `configd` is allowed to write to the partition containing the magic value. We discuss the use of USB tokens more in Section 6.4.5.
- **remove**. A USB token containing a magic value associated with `configd` has been removed.

To prevent applications from being able to modify c-locked file-system objects through modification of the kernel, the protection mechanism of Chapter 4 was implemented.

Backwards Compatibility

Because we implemented the backwards compatibility layer in the kernel (see Section 3), the kernel security module was extended to additionally send the **pipe** command to `configd`. **pipe** requests authorization from `configd` on requests received by the kernel to perform operations on c-locked file-system objects (i.e., the requests were not sent directly to `configd` by the application). The requests forwarded to `configd` by the kernel are *move*, *delete*, *link*, and *symlink*. `Configd` is responsible for performing permission checks on the request as if it came from some other user-space process. The id number of the request is sent back to the kernel through the use of the new command **id**. The response to the command is sent back to the kernel through the use of the new commands **done, fail, queued** and **unknown**, which are accompanied by an id number previously returned to the kernel by the **id** command. If the kernel receives a **done** response, it allows the application to perform the operation, otherwise the operation is denied. Other responses that can be received from `configd` indicate that the request has been rejected (*failed*), queued awaiting the user to insert a USB token (*queued*), or has failed to process due to some other error (*unknown*).

6.4.2 `Configd` Implementation

The `configd` prototype is a 12,500 line C++ application. While the primary purpose of writing `configd` is to encapsulate applications on disk, during the course of development several additional features were added. These include:

1. **A mounting module** which can decide whether a request to mount a file-system should be performed (it performs the operation, if allowed). To ensure that c-locked file-system objects remain accessible by applications they are associated with, we must ensure that a new file-system cannot be mounted over top of c-locked file-system objects. `Configd` does this by examining the *trusted.configd.nomount* file-system extended attribute. For directories which should not be mounted over (e.g. `/usr`), this extended attribute should be set on the directory inode. The mounting module also informs the kernel through the `configd` device node that requests to write to the underlying raw hard drive sectors should be denied, since allowing such requests would undermine the security of both c-locking and per-application restrictions enforced by `configd`.

2. **A modprobe module** which handles requests for the insertion or removal of kernel modules. In order to ensure that `configd` remains the chokepoint for restricting file-system objects on a per-application basis, root must not be allowed to install arbitrary code into the running kernel. While the prototype version of this module currently accepts all requests, kernel module loading can be easily restricted based on a number of criteria, including whether the module is signed with a recognizable key [96] (such an approach would not require a complete PKI infrastructure).

6.4.3 Debian Package Manager Modifications

Because the Debian package manager (`dpkg`) is responsible for performing most configuration changes on a system, we integrated `dpkg` with `configd`. While other methods of restricting configuration changes at a per-application level may choose to totally replace `dpkg`, augmenting `dpkg` to communicate with `configd` was suitable for our prototype (`configd` had the final say as to whether all requests for operations on c-locked files received from either the kernel or `dpkg` are allowed). The modified `dpkg` informed `configd` about each package which was being installed or upgraded, and also marked as c-locked every file installed when upgrading or installing a package, regardless of whether the file was previously c-locked.

The Debian package manager also keeps track of which files are associated with which packages. While we have not utilized it in our prototype implementation, this tie between files and packages can be used in other protection mechanisms which restrict configuration changes on a per-application level. We discuss several such approaches in Section 6.2. Because it is considered an error on Debian to have a file belonging to two unrelated packages [75], Debian's package approach lends itself nicely to a clean separation between applications.

In our prototype, the Debian package manager and surrounding infrastructure was responsible for preventing one application from assuming the name of another (and hence being able to modify the files associated with the second package). Such an approach depends on the security of the packaging system in Debian, which although reasonably secure against intrusion, is not perfect [31]. To reduce the dependence on Debian to keep packages from replacing other un-related packages in the archive, bin-locking (as discussed in Chapter 5) can be applied at the package level (as discussed in Section 5.2.2 and 5.7.1). A package is typically signed with a private key held by the developer. Any new version of that package is allowed to replace the installed one if it is signed with a private key verifiable with the corresponding public key contained in the currently installed package. In this way, application updates are restricted to those software authors holding the private signing key. Unless two applications are written by the same developer, and assuming that private keys are not generally shared between developers, the two applications are unlikely to have any identical keys and hence will not be able to modify each other. In using this approach for Debian packages, we can eliminate the risks associated with relying on the Debian packaging team to properly keep packages distinct [31].

6.4.4 Allowing Scripts During Package Install

In Section 6.3.3, rule 7 states that if c-locked file *A* is associated with package *PKG* and is not associated with any other package installed on the system, then install scripts associated with package *PKG* may modify *A*. In order to enforce that an install script contained in a package may only modify files on the system associated with that package (and no other package), we implemented an ability to run install scripts within `configd`. The approach closely parallels the approach used by GoboLinux for installing applications (see Section 6.2.1). The following procedure was followed for running install scripts:

1. An unused user ID (*UID*) is allocated by `configd`. While `configd` examined `/etc/passwd` in our prototype, different methods may be required when using alternate approaches for user account control.

2. For each file associated with package *PKG* (and not associated with any other package installed on the system), the owner of the file is recorded by `configd` and then changed to *UID*.

3. The script is run as user *UID* by `configd`. In running the script as a user who only has permission to modify files associated with the package, other applications on the system are protected by the standard access controls on the system.

4. For each file associated with the package *PKG*, the *UID* is changed back to the value stored in step 2, unless the user owning the file has been changed by the script.

5. The *UID* allocated in step 1 is freed.

The approach allows an install script associated with package *PKG* to modify any file associated with the same package, but does not allow the install script to modify any file associated with any other package installed on the system. It achieves the design goal of restricting file-system modifications such that one application can not modify the file-system objects associated with another application installed on the system. Indeed, simply implementing the approach into `dpkg` without fully implementing `configd` has benefits over the current approach of allowing unrestricted access to the file-system by install scripts.

In Linux, `ldconfig` is responsible for configuring the dynamic linker run-time bindings [1]. During testing of our prototype, scripts occasionally ran `ldconfig`, which in turn attempted to update `/etc/ld.so.cache`. This presents a problem, as `/etc/ld.so.cache` is not associated with the same package as the script, and hence modifications to the file will be denied. To compensate for this, `configd` runs `ldconfig` after the script completes with a *UID* that only has permission to write to `/etc/ld.so.cache`. Our prototype currently does not support the updating of symbolic links by `ldconfig`, relying on the package being installed to install the correct symbolic links. Should this become an issue in the future, `configd` can be updated to create symbolic links, similar to GoboLinux's `SymlinkProgram` (see Section 6.2.1).

The use of `fakeroot`[5] would provide a more comprehensive approach for allowing installation scripts to execute while not exposing the system to arbitrary file-system modifications. Using `fakeroot` allows `configd` to better emulate the permissions currently given to an install script without `configd` running. In the Linux environment, `fakeroot` is a program commonly used to make an application believe it has root privileges. The operations performed while running under `fakeroot` are recorded, and can be processed by `configd` with the goal of being able to replay the allowed operations later. This approach of recording operations and replaying them later is currently used when creating a distribution package from a source archive. A similar approach has also been used in other secure software installation systems [175].

[5] `http://fakeroot.alioth.debian.org/`

6.4.5 Handling of Operations not Automatically Allowed

While the prototype left enabled the option of querying the user for operations not allowed by other rules discussed in Section 6.3.3, we believe that this rule can be disabled when deploying `configd` to non-expert user systems, causing `configd` to reject of any file-system operation not allowed by the other rules. During the process of applying security updates to all packages modified between November 12th, 2009 and May 6th, 2010 (a total of $\sim$ 100 packages), we were not queried by `configd` at all about modifications to the file-system.

USB Tokens

The prototype must prevent race conditions, as well as having malware interfere with `configd`'s attempts to query the expert user about changes to c-locked file-system objects. The solution we adopted was to use a USB memory token tied to `configd` (i.e., it contains a special partition which is recognized by `configd`). When the user wishes to authorize changes to one or more c-locked file-system objects, a recognized USB token is inserted. The kernel then suspends all processes other than `configd` to prevent race conditions and tampering with the `configd` user interface. `configd` queries the user about any queued c-locked change requests. In relying on the physical insertion of a hardware token as a method to tie the user to a configuration change request, we borrow from the work of Butler et al. [30].

6.5 Evaluation of Prototype

In this section, we evaluate the prototype implementation. We discuss its performance, resistance to current malware, and experiences with using the system.

6.5.1 Performance Evaluation

To test the performance of our kernel modifications on file-system intensive day-to-day operations, we performed a complete compile of the Linux 2.6.31.5 kernel. We unpacked, configured (`make allmodconfig`), compiled, and removed the directory tree containing the compiled kernel. We chose a kernel unzip, compile, and removal because of the number of required disk operations, heavily exercising the file-system as well as our prototype c-locking Linux security module. We ran the test on two different 2.6.28.7 Linux kernels. The first test was with c-locking support not compiled in, and averaged 158 minutes and 52 seconds over three runs. The second timing was performed with c-locking enabled and `configd` running, and averaged 166 minutes and 34 seconds over three runs. Both tests were run on the same Pentium 4 2.8GHz with 1Gb of RAM. Over the three test runs, the average increased run time for the compile with c-locking enforcement enabled was 4.8%. For day to day operations which do not involve heavy file-system activity, we expect the overhead of `configd` to be well under 4.8%. We also expect alternate implementations of `configd` functionality would produce comparable results.

6.5.2 Verification that Application Encapsulation is Enforced

Malware frequently modifies file-system objects not directly associated with the malware itself (e.g., replacing *ls*). This provides an appropriate test case for the proposed restriction of configuration changes on a per-application basis during install. Under the new root file-system restrictions, the test is to verify that applications (malicious or other) running with root privileges cannot modify other applications file-system objects.

To test how well the mechanism presented in this chapter protects a system when exposed to malware, we became root on a system with `configd` running and kernel protections enabled. We then attempted to run six different Linux rootkit installers.[6] Linux rootkits can be grouped into two categories: those that use some method to gain access to kernel memory, installing themselves in the running kernel and then operating at kernel level, hiding their actions from even root processes; and rootkits that replace core system binaries that are often used by the root user in examining a system. Using the six representative rootkits, we confirmed that the installer failed to gain access to the kernel because of disabled write access to `/dev/kmem` (which would otherwise undermine `configd`), and that `configd` works as expected (i.e., file-system changes possible by malware are restricted to prevent other applications from being modified). That the rootkits failed to gain access to the kernel was verified by examining errors returned by the rootkit installer when attempting the install. The integrity of other applications file-system objects was verified through comparing cryptographic hashes using Tripwire [87].

We selected six representative Linux rootkits, two that modify the kernel and four that replace system binaries. Both kernel-based rootkits (`suckit2`

[6]All Linux rootkits tested were from `http://packetstormsecurity.org/UNIX/penetration/rootkits/`

and mood-nt) failed to install because of disabled write access to /dev/kmem in the prototype's modified kernel. The mood-nt kernel based rootkit which we tested also attempted to replace /bin/init. The attempt was denied by the prototype[7] because /bin/init is part of a different application on the system. The four binary replacement rootkits (ARK 1.0.1, cb-r00tkit, dica, and Linux Rootkit 5) all resulted in file-system operations which were denied because they attempted to either replace or delete core system binaries (e.g., ls, netstat, top, and ps). The core system binaries installed belong to applications other than the rootkits and hence changes to them by the rootkit installer were disallowed by our prototype.

6.5.3 Effects of Typical System Use

To test how well configd can be deployed on a pre-existing system, we introduced configd into a Debian desktop installation. We first installed an unmodified Debian Lenny (v5.0) distribution, including the KDE graphical desktop environment, onto a desktop. We then installed the configd daemon and modified kernel, which enforces c-locked file-system restrictions. At this point, any package updates on the system (including installs, re-installs, and upgrades) would result in the associated files being marked c-locked, and hence protected by configd. We then proceeded to re-install all packages that were already present on the system. In performing a re-install, all files associated with a package became marked as c-locked and hence protected by the combination of configd and the modified kernel.

Using this base system, where all package files are marked as c-locked, configd is running, and kernel protections are enabled, we then proceeded to do further testing of the prototype using our desktop install. To test how well the configd prototype accommodates package upgrades, we installed all security updates made available for our desktop install between the days of November 12th, 2009 and May 6th, 2010 (a total of $\sim$ 100 packages), we were not queried by configd at all about modifications to the file-system, we did not see any errors displayed during the upgrade process, and none of the packages appeared to be broken by the upgrades (i.e., the system still ran as expected). To test how well the prototype configd accommodates re-installs, we reinstalled all packages on the desktop system (a total of 897 packages). Again, during the process we were not queried about changes to the system configuration, we did not see any errors displayed during the process, and the system appeared to be

[7]In our prototype, the denial was performed by the prototype user, but would be performed automatically if the user-interface is disabled. The rootkit install was not able to put up a fake configd prompt because of protections discussed in Section 6.4.5.

fully functional after the reinstall (i.e., we could still log into KDE, browse the web, read e-mail, and watch videos).

As a final test, we installed several (new) desktop applications (each consisting of multiple packages), including Inkscape (`http://www.inkscape.org/`), Pidgin (`http://www.pidgin.im/`), and Digikam (`http://www.digikam.org/`). The new files installed during the application install were marked as c-locked, there were no errors displayed by the package installer, and the application ran correctly after install. The sequence of tests verified that the prototype system is capable of supporting the install of new packages, as well as the upgrade and re-install of already-installed packages.

6.5.4 Effects on Root of Restricting Configuration Changes

Because root no longer has permission to modify arbitrary files on disk, any configuration changes performed directly by the root user will be potentially be disallowed by `configd` or a related per-application encapsulation mechanism (configuration changes performed during install by the related install script are easily allowed; see Section 6.2). In the prototype system, configuration changes performed by the physical user acting as root ended up being approved, since the person modifying the application configuration and the person approving the change when queried by `configd` are one and the same. Because any updates to configuration require an individual acting as root to perform them, we believe the extra step of the same individual verifying the change before it is propagated to disk to be minor (e.g., the user updates the configuration file and then authorizes that the update be written to disk when prompted by `configd`). Indeed, we can avoid querying the user about configuration changes if the implementation of `configd` exports a user interface for performing changes (e.g., by providing a text editor). Such an ability does not detract from the security of the system because applications still do not gain the ability to write to file-system objects belonging to other applications.

6.5.5 Benefits over Android or GoboLinux

While the approach draws from both GoboLinux version 014.01 and Android version 2.0, there are several significant improvements over each approach (see Section 6.2). GoboLinux does not cleanly support upgrades, and hence each version of an application will be installed alongside the previous versions. A limitation with Android is that it did not support install scripts, and it relies on a permission model where each application is assigned its own account.

`Configd` takes the benefits of both (GoboLinux being able to run install scripts, and Android being able to support upgrades) and combines them, resulting in a system which supports upgrades and works with traditional binary packages and file-system layout.

6.6 Related Work

Here we discuss selected related work, beyond that discussed in Section 6.1.

6.6.1 Secure Software Installation

Venkatakrishnan et al. [175] proposed RPMShield, a system where actions which will be performed during install of a package are presented to the administrator for verification and then all install actions are logged. RPMShield concentrates on install time, not preventing already-installed applications from modifying the system if they are run as root. While `configd` focuses on encapsulating an application's file-system objects, RPMShield focuses on allowing the system administrator to examine and approve the actions which will be performed during install.

Kato et al. [80] proposed SoftwarePot, an approach where each application is encapsulated in its own sandbox, with mediated access to the file-system and network. Shared files are accessed by mapping sandbox-specific file-names to global file-system objects. The mapping between sandbox-specific files and global file-system objects requires additional information not currently distributed with an application package. SoftwarePot encountered a 21% overhead in execution time, while `configd` encountered a 4.8% overhead.

Sun et al. [163] proposed grouping applications into two categories, those which are trusted, and those which are not. All untrusted applications are installed inside a common sandbox, while trusted applications are not. The approach relies on the ability to always properly identify and classify malicious applications as untrusted. It does not prevent trusted applications from modifying file-system objects related to other trusted applications (and indeed, untrusted applications can modify file-system objects related to other untrusted applications). `Configd`, in contrast, does not distinguish between trusted and untrusted applications, treating all applications equally and restricting the modifications which can be performed on file-system objects.

6.6.2 Smart Phones

Those developing smart phones learnt from the instability and malware problems in the desktop space. Instead of segregating users (since each device is assumed to be used by a single user), they took the direction of segregating applications. Smart phone architectures also disabled the granting of root permission to applications on the system. Smart phone vendors have long retained strong control of their devices, having important reasons for limiting the damage an application could do to (or with) the device (e.g., due to strict broadcasting regulations).

On the iPhone platform, each application must be signed by Apple before being loaded onto the device. This approach assumes Apple will be able to properly vet all applications before allowing them to be loaded onto the device, an assumption which has proven risky (indeed, Apple now has a mechanism for disabling malicious applications which happen to slip through [85]). The approach also has scalability drawbacks.

The encapsulation approach taken by the Android smart phone is discussed in Section 6.2.2.

6.6.3 Rootkit Resistant Disks

Butler et al. [30] proposed a method where regions of disk were marked as requiring a specific USB key to be inserted before they could be updated. The approach works at the block level, underneath the file-system. Blocks on disk become marked as associated with a USB key when they are updated while the key is installed. In their approach, the user is involved in differentiating between when a system should be used to perform day-to-day operations and when the system is being configured. This separation, however, does not carry over into isolating day-to-day and configuration operations. Because the protection mechanism is implemented underneath the operating system at the block level, applications used for performing day-to-day operations continue to run (and even inherit configuration permission) when the user inserts a USB key indicating they want to change the configuration of the system. The Butler et al. approach of attempting to restrict configuration operations closely parallels the first step in our limiting the potential for abuse in root file-system privilege – the separation of configuration from normal day-to-day activities performed on a system.

The approach taken by Butler et al. attempts to further minimize the potential for abuse through the use of multiple USB keys, but does not reach an application level granularity. Indeed, they suggest using different tokens for different roles (e.g., one token associated with all binaries and another associated with all configuration files) [30].

6.6.4 Other Related Work

The splitting of root privilege is a common technique for limiting abuse in areas other than file-system control. Techniques such as capabilities [90, 64] and fine-grained access control [102] also split up root, but focus mainly on installed applications, not the installers themselves.

In the past few years, virtual machines (VMs) have started to become much more popular in server environments, to allow a single machine to run multiple instances of an operating system [13]. As part of the popularization of virtual environments, the opportunity to introduce additional segregation between applications has arisen. Each VM instance runs its own instance of an operating system and is assigned its own file-system and display. A typical setup involving virtual machines still groups many related applications together in a single VM. In this chapter, we focus on a method for dividing up root file-system privilege to prevent abuse by applications running on the same instance of the operating system, regardless of whether or not that OS happens to be running in a virtual machine. While the practice of an ordinary user installing applications into their home directory avoids root entirely, the application's file-system objects are not protected against modification by other applications the user installs (or indeed, any application the user happens to run).

Ioannidis et al. [70] introduced the concept of sub-operating systems, marking each file with a label indicating where it came from. These labels restrict what data files an application is allowed to access at a finer granularity than the user. Sub-OS does not explicitly tackle limiting the abuse of root file-system privilege during the process of installing, upgrading, and removing an application. Polaris [160] likewise focuses on application data, restricting what user files an application can access based on user interaction with the window manager.

Fine-grained access control systems, such as SELinux [102] and those implemented by Solaris [28], restrict an application's permission based on the labels assigned to both that application and the resources it wishes to use. These systems have the potential to split up root file-system privilege, parallelling the approach used herein. Traditionally, however, such policies have focused on run time system state (i.e., when the system is being used for day-to-day activities) as opposed to installers and related file-system configuration operations. In Linux, the default SELinux security policy has `dpkg` being granted write access to almost every file on the system [36]. Other systems, such as AppArmor

[18], appear to work best when new applications are not being installed or upgraded. In the current environment where applications are routinely upgraded, not supporting installs or upgrades is a problem.

While projects such as DTE-enhanced UNIX [180] and XENIX [167] restrict the privileges of root (including root's ability to configure the system), we are unaware of any such privilege systems designed to restrict configuration changes during install, upgrade, or uninstall (i.e., it seems most installers can still be run with such privileges, again still having full access to all file-system objects on disk). For systems using the OpenBSD `schg` [93] and ext2 immutable [169] flags, any application can be given the ability to change an immutable file – the user can simply be asked to run an application after acquiring sufficient configuration privileges. SVFS [195] protects files on disk but is susceptible to the same problem of inadequate control over installation applications.

There have been many attempts to detect malicious modifications to system configuration. Windows file protection (WFP) [37, 113] maintains a database of specific files which are protected, along with digital signatures of them. WFP is designed, however, to protect against a non-malicious end-user, preventing only accidental system modification. Pennington et al. [130] proposed implementing an intrusion detection system in the storage device to detect suspicious modifications. Strunk et al. [161] proposed logging all file modifications for a period of time to assist in the recovery after malware infection. Tripwire [87] maintains cryptographic hashes of all files in an attempt to detect modifications. All these proposals detect modifications after the fact. Applications such as registry watchers [53] and clean uninstallers [115] attempt to either detect or revert changes made to a system by an application installer. These systems similarly don't prevent changes in system configuration.

The separation of configuration privileges as proposed in this chapter prevents installers from making unauthorized changes to system state, leading to a proactive rather than reactive approach to limiting system configuration changes. Package managers [31] and the Microsoft installer [119] both limit system configuration actions allowed by packages designed for their system, but do not prevent applications from simply providing their own installer (or install script), bypassing the limits enforced by the package manager.

6.7 Final Remarks

The approach discussed in this chapter provides a method for restricting modifications to an application on disk. While updates to the application are allowed, the ability for an application to modify the file-system objects associated with another application are not allowed. These restrictions bring much-needed additional security to desktops in a world where different applications all share the file-system, and yet may not be trustworthy themselves. The approach does not rely on the end-user for enforcement, protects all applications installed, and is implemented by a guardian (the OS and `configd` developer). It therefore follows the thesis goal of providing a guardian enforced mandatory access control mechanism which can be deployed.

7 Summary and Concluding Remarks

In this thesis, we have proposed four new mandatory access control mechanisms. Each of these mechanisms was designed to be set by a guardian who is able to make security related decisions. In each approach, the application developers are, by design, not able to directly take control of policy decisions. Furthermore, end users need not themselves police the mechanism. In not requiring an expert user, we believe the mechanisms will be deployable to a wide audience.

7.1 A Summary of the Protection Mechanisms

7.1.1 SOMA

Many JavaScript-based attacks require that compromised web pages communicate with attacker-controlled web servers. The joint work SOMA restricts cross-domain communication to a web page's originating server and other servers that mutually approve of the cross-site communication. By preventing unapproved cross-domain communication, attacks such as cross-site scripting and cross-site request forgery can be blocked.

SOMA imposes no configuration or usage burden on end users. The policy is set by guardians, namely the administrators of sensitive web servers and web browser developers. The changes required by SOMA are easy for server administrators to understand, giving them a chance to specify what sites can interact with their content. SOMA is also incrementally deployable with incremental benefit. The limitations of SOMA include that it is not able to restrict communication between JavaScript functions loaded by the browser, it requires buy-in from browser vendors and site administrators, and it is only able to restrict traffic for web applications which rely on the browser.

7.1.2 Limiting Privileged Processor Permission

The current lack of protection between root level user control and privileged processor control on a system leads to a situation where it is possible for applications to bypass restrictions enforced on user space processes (even those run with root privileges). As an example, the current ability to prevent root from accessing a FUSE based file-system [57] can be bypassed by root altering the kernel. Root can also alter the page-table mapping of arbitrary processes by running code with privileged processor control.

The overly permissive permissions granted to root has a negative effect on the system. Both malware and legitimate user space processes have sufficient permission to negatively affect the system. For example, the ability to perform writes to the underlying hard drive on a mounted file-system can lead to users damaging to their own file-systems by corrupting critical system files, even though the user is using a non-malicious user space application.

The approaches detailed in this thesis for restricting the ability to run code with privileged processor control help prevent subversion of mandatory access control policies imposed by the kernel on user space processes – including pre-existing approaches such as SELinux [102], and AppArmor [18], as well as new mechanisms proposed in this thesis such as bin-locking (Chapter 5) and `configd` (Chapter 6). The restrictions include locking down raw writes to mounted file-systems and swap, restricting access to kernel memory device files, preventing arbitrary kernel modules from being loaded, and preventing arbitrary modification of the boot loader, kernel, and kernel modules on disk. While some of the restrictions are previously known and have been deployed by Microsoft on Windows, we focus our analysis on Linux by evaluating a prototype implementation. The implementation discussed in Chapter 4 has a negligible impact on end-user day-to-day use and defends against current malware rootkits.

7.1.3 Bin-Locking

When binaries are being installed, the current (almost universal) situation is that the installer has write access to essentially the entire file-system – far too coarse a granularity, from a security-oriented perspective. To address this, we presented bin-locking, a mechanism based on digitally signing software and extending the kernel to protect binaries on disk against modification by unauthorized software. One of the key features not widely addressed by previous file protection schemes (to our knowledge) is built-in support for software application upgrades. With many applications now receiving regular patches, dealing with upgrades in a smooth and non-intrusive manner is important. The proposed system enforces a separation between binary files belonging to different applications. Even with privileges sufficient to install an application, binary files belonging to one application cannot be modified by an application created by a different developer. While we do not restrict the ability to bin-lock binaries to certain vendors, we suspect that the vendors most interested in the capabilities offered by bin-locking may be those who develop or provide system monitoring utilities and crucial services.

7.1.4 `Configd`

`Configd` is a framework for restricting configuration changes, including kernel extensions and an associated user-space daemon. The status quo on end-user operating systems is dangerous – giving every application full access to the file-system during install, upgrade, and removal. We provide the foundation of a full solution including concept, architecture, design, and prototype implementation proving out the design. The design provides a mechanism for reducing root abuse of file-system privileges without breaking normal software operation.

Our end-goal is to eliminate the property whereby every process running with root privilege can change arbitrary files on disk, as this is commonly abused by malware on current desktop operating systems. A necessary part of this involves restricting the ability for applications to modify each other's file-system objects on disk. Our proposal mitigates the security risks associated with install mechanisms in common use today, wherein software installers (typically downloaded from the Internet) are run as root. The problem addressed herein is long-standing. While the proposal is specific to Linux, we see no reason why a similar approach could not be used on Windows.

7.2 Revisiting Thesis Objectives

As stated in Chapter 1, this thesis focuses on mandatory access control mechanisms which are set by a guardian. All mechanisms introduced in this thesis allow applications to continue to share resources, including the file-system and network. At the same time, the mechanisms introduced enforce a greater isolation of applications, preventing a number of the attacks which have become common place in both the desktop and web environments.

We met the constraints stated in the thesis hypothesis (on page 4) by designing each of the mandatory access control mechanisms so that they required little end-user or developer security expertise, and result in better overall application security. Each of the four mechanisms discussed was prototyped and tested to confirm that it could be used on current real-world systems. This included testing backwards compatibility (when appropriate), and verifying that the overhead of each mechanism was reasonable.

The mechanisms introduced provide new approaches for restricting the abilities of applications in an effort to better protect software systems against malware. The introduction of new kernel protections in Linux, along with bin-locking and `configd`, provide three guardian-based access control mechanisms positively answering the thesis question. SOMA provides a guardian-based access control mechanism which further supports our hypothesis. The mechanisms provided answer the stated question, providing four representative mechanisms which fall under the category of MAC policies being enforced by a guardian. We believe MAC policies set by a guardian are an important subset, choosing to formally recognize them. The mechanisms we discussed are designed to be implemented by the operating system developer. SOMA is designed to be implemented by the browser developer. By publishing these mechanisms, we draw attention to the guardian subset of mandatory access control policies, satisfying the second goal of our work (on page 4).

7.2.1 Overall Insights

In this section, we discuss several insights gleaned from developing the mandatory access control mechanisms discussed in this thesis. While our insights and personal views have arisen from our experience, rather than in all cases being directly supported by specific scientific experimentation, we nevertheless offer them to stimulate further discussion and exploration.

1. We believe that the guardian based access control mechanisms most acceptable to both application developers and users have characteristics common in many other domains, including parenting. Such policies involve both tenderness and firmness towards both users and developers. While we have no proof for such a statement, we believe that environments which are overly restrictive are likely to be disliked by either application developers or users. Furthermore, environments which are too

loosely constrained have led to the current situation in which malware is prevalent on the desktop and in the web environment.

2. We believe that the approach of not depending on application developers for security enforcement is the only realistic choice to make. Because application developers have various skill sets, it is not prudent to assume that they will all be capable of writing secure software. Furthermore, some application developers actively try to subvert systems by writing malware. To depend on the user to perform the task of security police appears also to be an unwise decision. The vast majority of computer users are not security experts. The approach taken in this thesis is to develop security mechanisms which are designed to be set by a knowledgeable guardian, one well-versed in security and capable of making decisions which benefit both legitimate application developers and users. We believe that in the past, too much trust has been placed in the developer's ability to security design their applications. We suspect this assumption of trust in the developer lead to a scenario where alternate approaches for increasing security were not actively pursued.

3. Computers were initially developed for use by very technically minded people in solving specific tasks [45]. Developers initially had complete freedom and flexibility in most of their design choices. Developers argue against limitations, preferring instead to retain complete freedom. This resistance from developers, however, is based on two different arguments, which we believe are non-overlapping. The first is the argument that the developer, as a user, will loose control over the ability to reconfigure the device they own. On this point, we choose not to prevent the owner of the hardware device from ultimately subverting the mandatory access control policies enforced in software. The other point is whether the developer should be allowed to have arbitrary control over other's devices through the software they develop. On this point, we take the side of not allowing the developer total control over other's devices, since malware commonly takes advantage of exactly this ability. In this thesis, we attempt to take away some of the freedom of developers when designing software for other's devices. We attempt to do this without offending too many other parties (i.e., without too many other negative impacts). We do so by creating new MAC policies which are set by a guardian, avoiding many end-user usability problems.

4. The mandatory access controls discussed in this thesis focused on areas which are often exploited by malware, but less seldom used by legitimate software. We chose to focus on exactly these areas in developing manda-

tory access control policies which can be set by a guardian. Should we have chosen to implement a more restrictive policy in areas commonly used by legitimate software, we believe we would have encountered significant usability problems. Indeed, many of the guardian based policies which have already been deployed (including the JavaScript same origin policy and execute disable) provide significant benefits while having acceptable usability by a non-expert. We believe there are still gains to be made in examining features made available by the application environment and commonly used by malware but not by legitimate software.

5. Both bin-locking and `configd` took advantage of the common developer paradigm of code reuse [58]. It is typically considered bad practice for a developer to re-implement someone else's code. One consequence of the paradigm of code reuse is that an individual shared library on disk is not likely to be created by multiple independent parties, a fact that bin-locking and `configd` used in their design. We believe there may be other guardian set policies which can be created by examining generally accepted developer design patterns.

7.3 Open Problems

In this thesis, we concentrated on preventing one software application from modifying (and thereby harming) another. In the web environment, the SOMA protection mechanism focused on the fetching of external content, since this is the most often used avenue through which attacks are performed. SOMA did not concentrate on those operations performed which do not require a fetch of content, or those operations which require fetching content from a server which has already been approved through the SOMA process. While the current approaches of further limiting the communication between web applications have been focused on giving the application developers the tools required to protect their applications, we do not believe there has been enough focus placed on how to ensure that these tools are used properly (or indeed, even at all). We therefore see opportunity for additional guardian based mandatory access controls which restrict communication between web applications (e.g., SOMA did not handle ad-syndication). We also see opportunity for additional research into generating improved fine-grained policies for web pages. (e.g., mashups [68, 74]).

Mutual approval, as discussed in Chapter 3 provides a mechanism for both parties to declare their approval for the sharing of information. While SOMA focused on applying this mutual approval mechanism to web applications, we see an opportunity for additional research into the application of such a policy outside the web context.

On the desktop, our solutions focused solely on preventing an application from obtaining privileged processor control, and restricting the ability to modify other applications' file-system objects on disk. This thesis did not focus on the problem of how to protect inter-process communication between applications. We believe there is the potential for additional guardian based mandatory access control policies to be developed which restrict the inter-process communication without placing significant additional assumptions on the abilities of either the end-user or application developer.

The desktop-based mandatory access control policies and related mechanisms discussed in this thesis are designed to limit the ability for applications to modify other applications file-system objects. Another problem which we do not address is that of a Trojan horse application. Such an application is self-contained, not needing to modify other applications on disk in order to carry out its nefarious activities. Bot-net clients which do not attempt to hide on the end-user desktop also fall into the category of not needing to modify other applications installed on the system in order to function. While this work results in malware being kept distinct from other applications on the file-system, it remains an open problem how to prevent application malware from being installed in the first place.

References

[1] *Linux Application Development*, chapter 8. Creating and Using Libraries. Addison-Wesley Professional, 2nd edition, Nov 2004.

[2] F. Adelstein, M. Stillerman, and D. Kozen. Malicious code detection for open firmware. In *Proc. 18th Annual Computer Security Applications Conference*, pages 403–412, Dec 2002.

[3] Adobe Systems Incorporated. External data not accessible outside a Macromedia Flash movie's domain. Technical Report tn_14213, Adobe Systems Incorporated, Feb 2006. `http://www.adobe.com/go/tn_14213`.

[4] Alexa top 500 sites. Web Page (viewed 14 Apr 2008). `http://www.alexa.com/site/ds/top_sites?ts_mode=global&lang=none`.

[5] Android developers. Web site (viewed 18 Nov 2009). `http://developer.android.com`.

[6] Apple, Inc. iPhone developer program license agreement, Jun 2008.

[7] Apple Inc. *iPhone Application Programming Guide*, Jan 2010. `http://developer.apple.com/iphone/library/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf`.

[8] A. Apvrille, D. Gordon, S. Hallyn, M. Pourzandi, and V. Roy. Digsig: Run-time authentication of binaries at kernel level. In *Proc. 18th USENIX Conference on System Administration (LISA)*, pages 59–66, Nov 2004.

[9] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *Proc. 18th IEEE Symposium on Security and Privacy*, pages 65–71, May 1997.

[10] M. Balduzzi, M. Egele, E. Kirda, D. Balzarotti, and C. Kruegel. A solution for the automated detection of clickjacking attacks. In *Proc. 5th ACM Symposium on Information, Computer and Communications Security*, Apr 2010.

[11] A. Baliga, X. Chen, and L. Iftode. Paladin: Automated detection and containment of rootkit attacks. Technical Report DCS-TR-593, Rutgers University Department of Computer Science, Jan 2006.

[12] A. Baliga, P. Kamat, and L. Iftode. Lurking in the shadows: Identifying systemic threats to kernel data. In *Proc. 28th IEEE Symposium on Security and Privacy*, pages 246–251, May 2007.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Oct 2003.

[14] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proc. 10th ACM Conference on Computer and Communication Security*, pages 281–289, Oct 2003.

[15] A. Barth, J. Caballero, and D. Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proc. 30th IEEE Symposium on Security and Privacy*, pages 360–371, May 2009.

[16] A. Barth, C. Jackson, and J. C. Mitchell. Robust defenses for cross-site request forgery. In *Proc. 15th ACM conference on Computer and Communications Security*, pages 75–88, Oct 2008.

[17] A. Barth, J. Weinberger, and D. Song. Cross-origin JavaScript capability leaks: Detection, exploitation, and defense. In *Proc. 18th USENIX Security Symposium*, Aug 2009.

[18] M. Bauer. Paranoid penguin: an introduction to Novell AppArmor. *Linux Journal*, 148:36,38,40–41, Aug 2006.

[19] A. Beautement, M. A. Sasse, and M. Wonham. The compliance budget: Managing security behaviour in organizations. In *Proc. 2008 New Security Paradigms Workshop*, pages 47–58, Sep 2008.

[20] D. Bell and L. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corporation, Mar 1973.

[21] D. Bell and L. LaPadula. Secure computer systems: Unified exposition and multics interpretation. Technical Report MTR-2545, Rev 1, MITRE Corporation, Mar 1975.

[22] A. Bellissimo, J. Burgess, and K. Fu. Secure software updates: Disappointments and new challenges. In *USENIX 2006 Workshop on Hot Topics in Security (HotSec 2006)*.

[23] K. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corporation, Apr 1977.

[24] M. Bishop. *Computer Security: Art and Science*. Addison Wesley, 2003.

[25] H. Bojinov, E. Bursztein, and D. Boneh. Xcs: Cross channel scripting and its impact on web applications. In *Proc. 16th ACM Conference on Computer and Communications Security*.

[26] D. Botta, R. Werlinger, A. Gangé, K. Beznosov, L. Iverson, S. Fels, and B. Fisher. Towards understanding IT security professionals and their tools. In *Proc. 3rd Symposium on Usable Privacy and Security*, Jul 2007.

[27] D. Brewer and M. Nash. The chinese wall security policy. In *Proc. 10th IEEE Symposium on Security and Privacy*, pages 206–214, May 1989.

[28] G. Brunette. Restricting service administration in the Solaris 10 operating system. Technical Report 819-2887-10, Sun Microsystems, 2005.

[29] bsign. Web site (viewed 22 Jan 2009). `http://packages.debian.org/lenny/bsign`.

[30] K. R. B. Butler, S. McLaughlin, and P. D. McDaniel. Rootkit-resistant disks. In *Proc. 15th ACM Conference on Computer and Communications Security*, pages 403–415, Oct 2008.

[31] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman. A look in the mirror: Attacks on package managers. In *Proc. 15th ACM Conference on Computer and Communications Security*, pages 565–574, Oct 2008.

[32] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systemic integrity checking. In *Proc. 16th ACM Conference on Computer and Communications Security*, Oct 2009.

[33] CERT. CERT advisory ca-2000-02: Malicious HTML tags embedded in client web requests. Technical Report CERT Advisory CA-2000-02, CERT, 2000. `http://www.cert.org/advisories/CA-2000-02.html`.

[34] S. Chen, D. Ross, and Y.-M. Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proc. 14th ACM Conference on Computer and Communications Security*, pages 2–11, Oct 2007.

[35] D. Clark and D. Wilson. A comparison of commercial and military security policies. In *Proc. 8th IEEE Symposium on Security and Privacy*, pages 184–194, May 1987.

[36] R. Coker. Re: [DSE-Dev] refpolicy: domains need access to the apt's pty and fifos. Mailing List Post, Mar 2008. `http://www.nsa.gov/research/selinux/list-archive/0803/25307.shtml`.

[37] J. Collake. Hacking Windows file protection. Web Page, 2007. `http://www.bitsum.com/aboutwfp.asp`.

[38] Common Weakness Enumeration. *2010 CWE/SANS Top 25 Most Dangerous Programming Errors*, Apr 2010. `http://cwe.mitre.org/top25/archive/2010/2010_cwe_sans_top25.pdf`.

[39] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo*, pages 119–129, Jan 2000.

[40] R. S. Cox, J. G. Hansen, S. D. Gribble, and H. M. Levy. A safety-oriented platform for web applications. In *Proc. 27th IEEE Symposium on Security and Privacy*, pages 350–364, May 2006.

[41] crazylord. Playing with Windows /dev/(k)mem. In *Phrack*, volume 0x0b (0x3b), chapter 0x10. Jul 2002. `http://www.phrack.com/issues.html?issue=59&id=16`.

[42] D. A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, 1992.

[43] S. Dandamudi. *Guide to RISC processors for programmers and engineers*. Springer, 2005.

[44] G. Davida, Y. Desmedt, and B. Matt. Defending systems against viruses through cryptographic authentication. In *Proc. 10th IEEE Symposium on Security and Privacy*, pages 312–318, May 1989.

[45] D. W. Davies. The bombe - a remarkable logic machine. *Cryptologia*, 23(2):108–138, Apr 1999.

[46] D. Dean, E. Felten, and D. Wallach. Java security: From HotJava to Netscape and beyond. In *Proc. 17th IEEE Symposium on Security and Privacy*, pages 190–200, May 1996.

[47] *The Debian GNU/Linux FAQ: Chapter 8 - The Debian Package Management Tools*, 2008. `http://www.debian.org/doc/FAQ/ch-pkgtools.en.html`.

[48] S. DeDeo. Pagestats extension. Web Page, May 2006. `http://www.cs.wpi.edu/~cew/pagestats/`.

[49] D. E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(2):236–243, 1976.

[50] P. J. Denning. *Computers Under Attack: Intruders, Worms, and Viruses*, chapter 17. Computer Viruses, page 290. Addison Wesley, 1990.

[51] D. Dittrich. "root kits" and hiding files/directories/processes after a break-in. Web Page, 2002. `http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq`.

[52] B. Dolan-Gavitt, A. Srivastava, P. Traynor, and J. Giffin. Robust signatures for kernel data structures. In *Proc. 16th ACM Conference on Computer and Communications Security*, Oct 2009.

[53] EasyDesk Software. Registry watch. Web Page (viewed 23 Apr 2009). `http://www.easydesksoftware.com/regwatch.htm`.

[54] C. Ellison. RFC 2692: SPKI requirements. Technical report, Internet Engineering Task Force, Sep 1999. `http://www.ietf.org/rfc/rfc2692.txt`.

[55] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. In *IEEE Software*, number 1 in 19, pages 42–51. IEEE Computer Society, Jan 2002.

[56] "Digital Signature Standard", Federal Information Processing Standards Publication 186. Technical report, U.S. Department of Commerce/N.I.S.T., National Technical Information Service, 1994.

[57] Fuse: Filesystem in userspace. Web Page (viewed 5 Mar 2010). `http://fuse.sourceforge.net/`.

[58] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[59] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. 10th Network and Distributed Systems Security Symposium*, pages 191–206, Feb 2003.

[60] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications (confining the wily hacker). In *Proc. 6th USENIX Security Symposium*, Jul 1996.

[61] Google. Android developer guide. Developer Website, 2009. `http://developer.android.com/guide/publishing/app-signing.html`.

[62] J. B. Grizzard. *Towards Self-Healing Systems: Re-establishing Trust in Compromised Systems*. PhD thesis, Georgia Institute of Technology, 2006.

[63] J. Grossman and T. Niedzialkowski. Hacking intranet websites from the outside – JavaScript malware just got a lot more dangerous. In *Blackhat USA*, Aug 2006.

[64] S. E. Hallyn and A. G. Morgan. Linux capabilities: Making them work. In *Proc. Ottawa Linux Symposium*, Jul 2008.

[65] J. Heasman. Implementing and detecting a PCI rootkit. In *Blackhat DC*, Nov 2007.

[66] R. Hensing. DEP on Vista exposed! Blog Posting, Apr 2007. `http://blogs.technet.com/robert_hensing/archive/2007/04/04/dep-on-vista-explained.aspx`.

[67] G. Hoglund and J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[68] J. Howell, C. Jackson, H. Wang, and X. Fan. MashupOS: Operating system abstractions for client mashups. In *Proc. 11th USENIX Workshop on Hot Topics in Operating Systems*, May 2007.

[69] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*. Number 253668. Intel, Dec 2009.

[70] S. Ioannidis, S. M. Bellovin, and J. M. Smith. Sub-operating systems: a new approach to application security. In *Proc. 10th Workshop on ACM SIGOPS European Workshop*, pages 108–115, Jul 2002.

[71] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. In *Proc. 14th ACM Conference on Computer and Communications Security*, pages 421–431, Oct 2007.

[72] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting browsers from dns rebinding attacks. *ACM Transactions on the Web*, 3(1), 2009.

[73] C. Jackson, A. Bortz, D. Boneh, and J. C. Mitchell. Protecting browser state from web privacy attacks. In *Proc. 15th International Conference on World Wide Web*, pages 737–744, May 2006.

[74] C. Jackson and H. J. Wang. Subspace: secure cross-domain communication for web mashups. In *Proc. 16th International Conference on World Wide Web*, pages 611–62, May 2007.

[75] I. Jackson and C. Schwarz. *Debian Policy Manual*, 1998. `http://www.debian.org/doc/debian-policy/`.

[76] T. Jaeger, R. Sailer, and X. Zhang. Analyzing integrity protection in the SELinux example policy. In *Proc. 12th USENIX Security Symposium*, pages 59–74, Aug 2003.

[77] M. Jakobsson and Z. Ramzan. *Crimeware: Understanding New Attacks and Defenses*. Addison-Wesley Professional, 2008.

[78] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Proc. 2nd IEEE Conference on Security and Privacy in Communication Networks*, Aug 2006.

[79] C. Karlof, J. Tygar, D. Wagner, and U. Shankar. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proc. 14th ACM Conference on Computer and Communications Security*, pages 58–71, Oct 2007.

[80] K. Kato and Y. Oyama. Softwarepot: An encapsulated transferable file system for secure software circulation. In *Proc. of International Symposium on Software Security*, volume Lecture Notes in Computer Science 2609/2003, pages 217–224, 2003.

[81] B. Kauer. Oslo: Improving the security of trusted computing. In *Proc. 16th USENIX Security Symposium*, pages 229–237, Aug 2007.

[82] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proc. 10th ACM Conference on Computer and Communication Security*, pages 272–280, Oct 2003.

[83] K. Keahey, K. Doering, and I. Foster. From sandbox to playground: Dynamic virtual environments in the grid. In *Proc. Fifth IEEE/ACM International Workshop on Grid Computing*, pages 34–42, Nov 2004.

[84] J. Keith. *DOM Scripting: Web Design With JavaScript and the Document Object Model*, chapter 3. The Document Object Model. Springer-Verlag, 2005.

[85] A. Kim. Apple's ability to deactivate malicious app store apps. Web Page, 2008. `http://www.macrumors.com/2008/08/06/apples-ability-to-deactivate-malicious-app-store-apps/`.

[86] G. H. Kim and E. H. Spafford. Experiences with Tripwire: Using integrity checkers for intrusion detection. Technical Report CSD-TR-93-071, Purdue University, 1993.

[87] G. H. Kim and E. H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proc. 2nd ACM Conference on Computer and Communications Security*, pages 18–29, Oct 1994.

[88] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, and Y. Zhou. Designing and implementing malicious hardware. In *Proc. 1st USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Apr 2008.

[89] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A client-side solution for mitigating cross site scripting attacks. In *Proc. 21st ACM Symposium on Applied Computing*, pages 330–337, Apr 2006.

[90] A. Kjeldaas. Linux capability FAQ v0.1. Mailing List Post, Aug 1998. `http://lkml.indiana.edu/hypermail/linux/kernel/9808.1/0178.html`.

[91] D. V. Klein. Defending against the wily surfer — web-based attacks and defenses. In *Proc. 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, pages 9–21, Apr 1999.

[92] Knoppix Linux. Web Page (viewed 15 Dec 2008). `http://www.knoppix.net`.

[93] Y. Korff, P. Hope, and B. Potter. *Mastering FreeBSD and OpenBSD Security*, chapter 2.1.2. O'Reilly, 2005.

[94] D. Kristol and L. Montulli. RFC2109: HTTP state management mechanism. Technical report, Internet Engineering Task Force, Feb 1997. `http://www.ietf.org/rfc/rfc2109.txt`.

[95] D. Kristol and L. Montulli. RFC2965: HTTP state management mechanism. Technical report, Internet Engineering Task Force, Oct 2000. `http://www.ietf.org/rfc/rfc2965.txt`.

[96] G. Kroah-Hartman. Signed kernel modules. *Linux Journal*, 117:48–53, Jan 2004.

[97] I. Krsul and E. H. Spafford. Authorship analysis: Identifying the author of a program. *Computers & Security*, 16(3):233–257, 1997.

[98] C. Kruegel, W. Robertson, and G. Vigna. Detecting kernel-level rootkits through binary analysis. In *Proc. 20th Annual Computer Security Applications Conference*, pages 91–100, Dec 2004.

[99] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proc. 13th ACM Conference on Computer and Communications Security*, pages 221–234, Oct 2006.

[100] Programming language popularity. Web Page, Jan 2010. `http://langpop.com/`.

[101] Q. Liu, R. Safavi-Naini, and N. P. Sheppard. Digital rights management for content distribution. In *Proc. Australasian Information Security Workshop Conference on ACSW Frontiers*, volume 21, pages 49–58, 2003.

[102] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proc. FREENIX Track: USENIX Annual Technical Conference*, pages 29–42, Jun 2001.

[103] M. T. Louw and V. Venkatakrishnan. Blueprint: Robust prevention of cross-site scripting attacks for existing web browsers. In *Proc. 30th IEEE Symposium on Security and Privacy*, pages 331–346, May 2009.

[104] R. Love. *Linux Kernel Development*. Novell Press, second edition, 2005.

[105] G. Maone. NoScript - JavaScript/Java/Flash blocker for a safer Firefox experience! Web page (viewed 14 Apr 2008). `http://noscript.net/`.

[106] G. Maone. Hardening the web with NoScript. *;Login: The USENIX Magazine*, 34(6):21–27, 2009.

[107] B. McCarty. *SELinux: NSA's Open Source Security Enhanced Linux*. O'Reilly Media, Inc., 2004.

[108] M. K. McKusick. Running "fsck" in the background. In *Proc. 2nd USENIX BSD Conference*, pages 55–64, Feb 2002.

[109] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, fifth edition, 1996.

[110] Microsoft Corporation. Device\PhysicalMemory object. TechNet Article (viewed 20 Feb 2010). `http://technet.microsoft.com/en-us/library/cc787565(WS.10).aspx`.

[111] Microsoft Corporation. Mitigating cross-site scripting with HTTP-only cookies. MSDN Article (viewed 8 Feb 2010). `http://msdn.microsoft.com/en-us/library/ms533046.aspx`.

[112] Microsoft Corporation. A detailed description of the data execution prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003. Technical report, Microsoft Corporation, Sep 2006. `http://support.microsoft.com/kb/875352`.

[113] Microsoft Corporation. Description of the Windows file protection feature. Web Page, 2007. `http://support.microsoft.com/kb/222193`.

[114] Microsoft Corporation. *Digital Signatures for Kernel Modules on Systems Running Windows Vista*, Jul 2007. `http://www.microsoft.com/whdc/winlogo/drvsign/kmsigning.mspx`.

[115] Microsoft Corporation. Description of the windows installer cleanup utility. Technical Report Q290301, Microsoft Corporation, 2008. `http://support.microsoft.com/kb/290301`.

[116] Microsoft Corporation. WriteFileEx function. MSDN Article, Nov 2008. `http://msdn.microsoft.com/en-us/library/aa365748(VS.85).aspx`.

[117] C. Moock. *Essential ActionScript 3.0*, chapter 19. Flash Player Security Restrictions. O'Reilly Media, Inc., 1st edition edition, 2007.

[118] J. Morris. Have you driven an SELinux lately? In *Proc. Ottawa Linux Symposium*, Jul 2008.

[119] J. Moskowitz and D. Sanoy. *The Definitive Guide to Windows Installer Technology*. Realtimepublishers.com, 2002. `http://nexus.realtimepublishers.com/dgwit.php`.

[120] H. Muhammad. Compiling from source. Web Page (viewed 16 Feb 2010). `http://gobo.kundor.org/wiki/Compiling_From_Source`.

[121] H. Muhammad. The Unix tree rethought: an introduction to GoboLinux. Kuro5hin Article, May 2003. `http://gobolinux.org/index.php?page=k5`.

[122] H. Muhammad and A. Detsch. Uma nova proposta para a árvore de diretórios UNIX. In *Proceedings of the III WSL - Workshop em Software Livre*, 2002.

[123] D. Muthukumaran, A. Sawani, J. Schiffman, B. M. Jung, and T. Jaeger. Measuring integrity on mobile phone systems. In *Proc. 13th ACM Symposium on Access Control Models and Technologies*, pages 155–164, Jun 2008.

[124] R. Nolan and R. X. Tang. PC with multiple video-display refresh-rate configurations using active and default registers. United States Patent Application US2000/6049316, Apr 2000.

[125] S. Oaks. *Java Security*, chapter 12. Digital Signatures. O'Reilly Media, Inc., 2nd edition, May 2001.

[126] T. Oda, G. Wurster, P. van Oorschot, and A. Somayaji. SOMA: Mutual approval for included content in web pages. In *Proc. 15th ACM conference on Computer and Communications Security*, pages 89–98, Oct 2008.

[127] Y. K. Okuji. GNU GRUB. Web Page, Dec 2008. `http://www.gnu.org/software/grub/`.

[128] P. Padala. Playing with `ptrace`, part 1. *Linux Journal*, 103, Nov 2002.

[129] A. Pennarun, B. Allombert, and P. Reinholdtsen. Debian popularity contest. `http://popcon.debian.org/`.

[130] A. Pennington, J. Strunk, J. Griffin, C. Soules, G. Goodson, and G. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proc. 12th USENIX Security Symposium*, pages 137–151, Aug 2003.

[131] N. L. Petroni Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proc. 13th USENIX Security Symposium*, pages 179–194, Aug 2004.

[132] N. L. Petroni Jr., T. Fraser, A. Walters, and W. Arbaugh. An architecture for specification-based detection of semantic integrity violations in kernel dynamic data. In *Proc. 15th USENIX Security Symposium*, pages 289–304, Aug 2006.

[133] A. Pfiffer. *Reducing System Reboot Time With kexec*. Open Source Development Labs, Inc., Apr 2003.

[134] M. Pozzo and T. Gray. An approach to containing computer viruses. *Comuters & Security*, 6(4):321–331, 1987.

[135] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your `iFRAME`s point to us. In *Proc. 17th USENIX Security Symposium*, pages 1–15, Aug 2008.

[136] Red Hat, Inc. *Fedora Core 5 - Release Notes*, Feb 2006. `http://docs.fedoraproject.org/release-notes/fc5/release-notes-ISO/`.

[137] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proc. 27th IEEE Symposium on Security and Privacy*, pages 61–74, May 2006.

[138] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Transactions on the Web*, 1(3), 2007.

[139] R. Repasi and S. Clausen. Method and system to scan firmware for malware. United States Patent Application US2007/0277241 A1, Nov 2007.

[140] R. Riley, X. Jiang, and D. Xu. An architectural approach to preventing code injection attacks. In *Proc. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 30–40, Jun 2007.

[141] R. L. Rivest and B. Lampson. *A Simple Distributed Security Infrastructure*, Oct 1996. `http://people.csail.mit.edu/rivest/sdsi11.html`.

[142] A. Rubin and D. Geer. Mobile code security. *IEEE Journal on Internet Computing*, 2(6):30–34, 1998.

[143] N. Ruff. Windows memory forensics. *Journal in Computer Virology*, 4(2):83–100, May 2008.

[144] R. Russell, D. Quinlan, and C. Yeoh. *Filesystem Hierarchy Standard*. Filesystem Hierarchy Standard Group, 2.3 edition, Jan 2004. `http://www.pathname.com/fhs/`.

[145] J. Rutkowska. Subverting Vista kernel for fun and profit. In *Blackhat USA*, Aug 2006.

[146] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proc. 13th USENIX Security Symposium*, pages 223–238, Aug 2004.

[147] J. Schuh. Same-origin policy part 2: Server-provided policies? Web Page, Feb 2007. `http://taossa.com/index.php/2007/02/17/same-origin-proposal/`.

[148] sd and devik. Linux on-the-fly kernel patching without LKM. In *Phrack*, volume 0x0b (0x3a), chapter 0x07. Dec 2001. `http://www.phrack.org/issues.html?issue=58&id=7`.

[149] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Model-carrying code (MCC): a new paradigm for mobile-code security. In *Proc. 2001 New Security Paradigms Workshop*, pages 23–30, Sep 2001.

[150] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc w ithout function calls (on the x86). In *Proc. 14th ACM Conference on Computer and Communications Security*, pages 552–561, Oct 2007.

[151] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proc. 11th ACM Conference on Computer and Communications Security*, pages 298–307, Oct 2004.

[152] M. Sharif, W. Lee, and W. Cui. Secure in-VM monitoring using hardware virtualization. In *Proc. 16th ACM Conference on Computer and Communications Security*, Oct 2009.

[153] A. Siberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley, seventh edition, 2005.

[154] E. Skoudis and L. Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, 2004.

[155] S. Smalley, C. Vance, and W. Salamon. Implementing SELinux as a linux security module. Technical Report 01-043, NAI Labs, May 2002.

[156] S. Spainhour, E. Siever, and N. Patwardhan. *Perl in a Nutshell*, chapter 8.25. Benchmark. O'Reilly Media, Inc., 2nd edition, 2002.

[157] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, fourth edition, 2001.

[158] B. Sterne. Security/CSP. Web Page (viewed 7 Jan 2010). `https://wiki.mozilla.org/Security/CSP`.

[159] B. Sterne. Site security policy draft (version 0.2). Web Page, Jul 2008. `http://people.mozilla.org/~bsterne/site-security-policy/details.html`.

[160] M. Stiegler, A. H. Karp, K.-P. Yee, T. Close, and M. S. Miller. Polaris: virus-safe computing for Windows XP. *Communications of the ACM*, 49(9):83–88, 2006.

[161] J. Strunk, G. Goodson, M. Scheinholtz, C. Soules, and G. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proc. 4th USENIX Symposium on Operating Systems Design and Implementation*, Oct 2000.

[162] S. Sudre. Packagemaker how-to. Web Page (viewed 29 Oct 2009). `http://s.sudre.free.fr/Stuff/PackageMaker_Howto.html`.

[163] W. Sun, R. Sekar, Z. Liang, and V. N. Venkatakrishnan. Expanding malware defense by securing software installations. *Lecture Notes in Computer Science*, 5137/2008:164–185, 2008.

[164] L. Tauscher and S. Greenberg. How people revisit web pages: empirical findings and implications for the design of history systems. *International Journal of Human Computer Studies*, 47(1):97–137, 1997.

[165] The Open Web Application Security Project. *OWASP Top 10 - 2010: The Ten Most Critical Web Application Security Risks*, Apr 2010. `http://owasptop10.googlecode.com/files/OWASPTop10-2010.pdf`.

[166] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, version 1.2 edition, May 1995. `http://refspecs.freestandards.org/elf/elf.pdf`.

[167] Trusted Information Systems, Inc. Trusted XENIX version 3.0 final evaluation report. Technical Report CSC-EPL-92-001, National Computer Security Center, Apr 1992.

[168] T. Y. Ts'o and S. Tweedie. Planned extensions to the Linux ext2/ext3 filesystem. In *Proc. FREENIX Track: USENIX Annual Technical Conference*, pages 235–243, Jun 2002.

[169] C. Tyler. *Fedora Linux*, chapter 8.4. O'Reilly, 2007.

[170] Social engineering (trojan) via gnome-loook.org. Web Page (viewed 13 Feb 2010). `http://ubuntuforums.org/showthread.php?t=1349801`.

[171] A. van de Ven. [patch] NX (No eXecute) support for x86, 2.6.7-rc2-bk2. Mailing List Post, Jun 2004. `http://lkml.org/lkml/2004/6/2/238`.

[172] A. van de Ven. make /dev/kmem a config option. GIT Commit, Apr 2008. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6-stable.git;a=commit;h=b781ecb6a379f155568ef7093e38c6c1d857fe53`.

[173] A. van de Ven. x86: Introduce /dev/mem restrictions with a config option. GIT Commit, Apr 2008. `http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6-stable.git;a=commit;h=ae531c26c5c2a28ca1b35a75b39b3b256850f2c8`.

[174] L. van Doorn, G. Ballintign, and W. A. Arbaugh. Signed executables for Linux. Technical Report CS-TR-4259, University of Maryland, 2001.

[175] V. N. Venkatakrishnan, R. Sekar, T. Kamat, S. Tsipa, and Z. Liang. An approach for secure software installation. In *Proc. 16th USENIX Conference on System Administration (LISA)*, pages 219–226, Nov 2002.

[176] K. Vervloesem. Linux malware: an incident and some solutions. LWN.net Article, Dec 2009. `http://lwn.net/Articles/367024/`.

[177] S. Vidyaraman, M. Chandrasekaran, and S. Upadhyaya. Position: The user is the enemy. In *Proc. 2007 New Security Paradigms Workshop*, pages 75–80, Sep 2007.

[178] P. Vogt, F. Nentwich, N. Jovanovic, C. Kruegel, E. Kirda, and G. Vigna. Cross site scripting prevention with dynamic data tainting and static analysis. In *Proc. 14th Network and Distributed System Security Symposium*, pages 67–78, Feb 2007.

[179] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolatio. *ACM SIGOPS Operating System Review*, 27(5):203–216, 1993.

[180] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Sherman, and K. A. Oostendorp. Confining root programs with domain and type enforcement (DTE). In *Proc. 6th USENIX Security Symposium*, Jul 1996.

[181] H. J. Wang, X. Fan, C. Jackson, and J. Howell. Protection and communication abstractions for web browsers in MashupOS. *ACM SIGOPS Operating Systems Review*, 41(6):1–16, Dec 2007.

[182] Y.-M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proc. 35th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 368–377, Jun 2005.

[183] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proc. 16th ACM Conference on Computer and Communications Security*, Nov 2009.

[184] Web Application Security Consortium. *WASC Threat Classification*, Jan 2010. `http://projects.webappsec.org/f/WASC-TC-v2_0.pdf`.

[185] WebSense. Super Bowl XLI / Dolphin Stadium - security labs alert. Web Page, Feb 2007. `http://securitylabs.websense.com/content/Alerts/1346.aspx`.

[186] K. C. Wilbur and Y. Zhu. Click fraud. *Marketing Science*, 28(2):293–308, Mar 2009.

[187] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *Proc. 11th USENIX Security Symposium*, pages 17–31, Aug 2002.

[188] C. P. Wright and E. Zadok. Unionfs: Bringing filesystems together. *Linux Journal*, 128:24–29, Dec 2004.

[189] G. Wurster and P. van Oorschot. Self-signed executables: Restricting replacement of program binaries by malware. In *USENIX 2007 Workshop on Hot Topics in Security (HotSec 2007)*.

[190] G. Wurster and P. van Oorschot. The developer is the enemy. In *Proc. 2008 New Security Paradigms Workshop*, Sep 2008.

[191] G. Wurster and P. C. van Oorschot. System configuration as a privilege. In *USENIX 2009 Workshop on Hot Topics in Security (HotSec 2009)*.

[192] G. Wurster and P. C. van Oorschot. Towards reducing unauthorized modification of binary files. Technical Report TR-09-07, Carleton University, 2009.

[193] Z. Ye, S. Smith, and D. Anthony. Trusted paths for browsers. *ACM Transactions on Information and System Security*, 8 (2):153–186, 2005.

[194] M. Zalewski. Browser security handbook. Online Book (viewed 21 Feb 2010). `http://code.google.com/p/browsersec/`.

[195] X. Zhao, K. Borders, and A. Prakash. Towards protecting sensitive files in a compromised system. In *Proc. 3rd IEEE International Security in Storage Workshop*, pages 21–28, Dec 2005.

Index