CERIAS Tech Report 2009-36

Improving real-world access control systems by identifying the true origins of a request

by Ziqing Mao
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Ziqing Mao

Entitled Improving Real-World Access Control Systems by Identifying the True Origins of a Request

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Ninghui Li
Chair

Elisa Bertino

Dongyan Xu

Xiangyu Zhang

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Ninghui Li

Approved by: William J. Gorman 17 November, 2009
Head of the Graduate Program Date

Graduate School Form 20
(Revised 6/09)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation:
Improving Real-World Access Control Systems by Identifying the True Origins of a Request

For the degree of Doctor of Philosophy

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Ziqing Mao
Printed Name and Signature of Candidate

11/28/2009
Date (month/day/year)

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

IMPROVING REAL-WORLD ACCESS CONTROL SYSTEMS BY IDENTIFYING THE TRUE ORIGINS OF A REQUEST

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Ziqing Mao

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2009

Purdue University

West Lafayette, Indiana

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

To my dear dad.

ACKNOWLEDGMENTS

First and foremost, I offer my sincerest gratitude to my advisor, Prof. Ninghui Li, who has guided and supported me throughout my Ph.D. study with his patient, knowledge and wisdom whilst allowing me the room to work in my own way. He is always willing to help, both on academic development and on personal issues. One simply could not wish for a better or friendlier advisor, and I consider it a work of providence that I found him as my advisor and friend.

Hong Chen, for contributing extensively to the operating system security project and a wide variety of other assistances.

Dr. Shuo Chen, for mentoring me through the HTTPS security project and introducing me to the interesting area of web security.

I would also like to thank my thesis committee, Prof. Dongyan Xu, Prof. Xiangyu Zhang, Prof. Elisa Bertino and Prof. Sam Wagstaff for their direction, dedication, and invaluable advice for this dissertation.

I am also indebted to Prof. Cristina Nita-Rotaru, Tiancheng Li, Qun Ni and Qihua Wang. Many thanks for your kind help and support during my Ph.D. study.

My parents, Xuan Chen and Zuzhang Mao, and my sister, Yilan Mao, receive my deepest gratitude. I attribute the level of my Ph.D. degree to their encouragements and efforts and without them I would not be able to pursue a Ph.D. degree.

Last, but certainly not least, I thank my wife, Ling Tong, for her understanding and love. I would like to share this with her.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Mao Ziqing Ph.D., Purdue University, December 2009. Improving Real-World Access Control Systems by Identifying the True Origins of a Request. Major Professor: Ninghui Li.

Access control is the traditional center of gravity of computer security. In order to make correct access control decisions, a critical step is to identify the origins of an access request. The origins of a request are the principals who cause the request to be issued and the principals who affect the content of the request. Therefore, the origins are responsible for the request. The access control decision should be based on the permissions of the origins.

In this dissertation, we examined two real-world access control systems, operating system access control and browser access control. They are vulnerable to certain attacks because of their limitations in identifying the origins of a request. In particular, the discretionary access control (DAC) in the operating system is vulnerable to Trojan horses and vulnerability exploits, while the same origin policy (SoP) in the browser is vulnerable to the malicious proxy adversary against HTTPS and the cross-site request forgery attack. We proposed enhancements of both systems by identifying the true origins of a request. We discussed the design details, the prototype implementations, and the experimental evaluations of the enhancements.

1 INTRODUCTION

In computer systems the functionality of the access control is to control which principals (persons, processes, machines, ...) have access to which resources in the system — which files they can read, which programs they can execute, and how they share the data with other principals, and so on [1]. Access control is critical to the security of real-world computer systems. For example, the host security heavily relies on the access control enforced by the operating system. The Firewall, which is essentially an access control system, plays an important role in the security of an Enterprise network. In Database systems, the access control restricts which users are allowed to access which tables. In browsers, the access control enforces the same-origin-policy (SoP) model to ensure that user sessions with different websites are properly isolated.

The key functionality of an access control system is to perform the *access control check*. When an *access request* happens in the computer system, the access control check makes an *access control decision* based on the *access control policy* specified by the administrator. Typically, the access control decision is either *allow* or *deny*.

Traditionally, an access request is comprised of three components: the subject, the object and the access mode. The subject is the entity that issues the request. In computer systems, the subject is typically a piece of running code, e.g., a process in the operating system. The object is the resource the subject wants to access, e.g., a file in the operating system and a table in the database system. Examples of access modes include read, write, etc. Given a request, the access control check needs to determine whether the subject has the privilege to access the object in the access mode based on the policy.

However, in real-world access control systems, the policy does not grant the privileges to the subjects. Instead, the privileges are granted to principals in the policy. For example, in the operating system, privileges are granted to user accounts. In the

browser, privileges are granted to domains (e.g., https://www.bank.com). Therefore, when checking a request against the policy, the access control mechanism has to fill the gap between the subjects in the requests and the principals in the policy. To do that, we introduce the notion of *origins* of a request.

At any time, a subject is executing on the behalf of one or more principals. The origins of a request are defined as the set of principals on whose behalf the subject is executing at the time the request is issued. In this way, the origins of a request are actually the principals that cause the subject to issue the request and the origins should be responsible for the request. As a result, the access control check is to check the request against the privileges of the origins of the request.

To properly identify the origins of a request is essential to making a correct access control decision. Many real-world access control systems are vulnerable to certain attacks because of their limitations in identifying the origins. In particular, we target two real-world access control systems, operating system access control and browser access control. The discretionary access control (DAC) used in operating systems is vulnerable to Trojan horses and vulnerability exploits, whilst the same-origin policy (SoP) model used in browsers is vulnerable to the pretty-bad-proxy adversary against HTTPS and the cross-site request forgery attack. We show that these access control systems are vulnerable because they do not correctly identify the origins of the requests.

- First, in DAC the origin of a request is defined as the user account who invokes the subject process. However, if the program is malicious or contains vulnerabilities that have been exploited, it may perform malicious activities that are not intended by the invoker. As a result, DAC is vulnerable to the vulnerability exploits and Trojan horses.

- Second, the SoP model protects the integrity of web session whilst relying on the browser to correctly identify the origins of the web objects and documents. The HTTPS deployments in major browsers fail to identify the true origins of

web messages. They mistakenly accept the web messages transmitted over an insecure connection into an HTTPS session. As a result, a malicious man-in-the-middle is able to corrupt the integrity of an HTTPS session.

- Third, the cross-site request forgery attack is feasible because the SoP model does not restrict an HTTP request to carry the user's authentication token based on the origin of the request. As a result, a malicious website is able to forge a cross-site request addressing a sensitive website on the user's behalf, using the user's authentication token associated with the sensitive website.

Based on the analysis on the existing access control systems and their weaknesses, we propose enhancements to make them resilient to those attacks.

In this chapter, we first describe the basic access control models that are currently used in the operating system and the browser. Then we analyze why they are vulnerable to certain attacks. Finally, we introduce the basic idea of the proposed enhancements.

1.1 Operating System Access Control

The general goal of the operating system access control is to prevent the adversary from compromising the host system. Host compromise is one of the most serious computer security problems today. Computer worms propagate by first compromising vulnerable hosts and then propagate to other hosts. Compromised hosts may be organized under a common command and control infrastructure, forming botnets. Botnets can then be used for carrying out attacks, such as phishing, spamming, distributed denial of service, and so on. These threats can be partially dealt with at the network level using valuable technologies such as firewalls and network intrusion detection systems. However, to effectively solve the problem, one has to also deal with the root cause of these threats, namely, the vulnerability of end hosts. One key reason why hosts can be easily compromised is because the discretionary access

control mechanism in today's operating systems is insufficient for protecting hosts against vulnerability exploits and malicious software.

1.1.1 The Traditional Discretionary Access Control

Modern commercial-off-the-shelf (COTS) operating systems use Discretionary Access Control (DAC) to protect files and other operating system resources. According to the Trusted Computer System Evaluation Criteria (TCSEC) (often referred to as the Orange Book) [2], Discretionary Access Control is "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (unless restrained by mandatory access control)." It has been known since early 1970's that DAC is vulnerable to Trojan horses. A Trojan horse, or simply a Trojan, is a piece of malicious software that in addition to performing some apparently benign and useful actions, also performs hidden, malicious actions. Such Trojans may come from email attachments, programs downloaded from the Internet, or removable media such as USB thumb drives. By planting a Trojan, an attacker can get access to resources the attacker is not authorized under the DAC policy, and is often able to abuse such privileges to take over the host or to obtain private information. DAC is also vulnerable when one runs buggy programs that receive malicious inputs. For example, a network-facing server daemon may receive packets with mal-formed data, a web browser might visit malicious web pages, and a media player can read malformed data stored on a shared drive. An attacker can form the input to exploit the bugs in these programs and take over the processes running them, e.g., by injecting malicious code. In essence, a buggy program that takes malicious input can become a Trojan horse.

For existing DAC mechanisms to be effective in achieving the specified protection policies, one has to assume that *all* programs are benign (be functional as intended

and free of malicious side effects) and correct (won't be exploited by malicious inputs). This assumption does not hold in today's computing environments. This weakness of DAC is a key reason that today's computer hosts are easily compromised.

Even though DAC's weaknesses is widely known since early 1970's, DAC is today's dominant access control approach in operating systems. We believe that this is because DAC has some fundamental advantages when compared with mandatory access control (MAC). DAC is easy and intuitive (compared with MAC) for users to configure, many computer users are familiar with it, and the discretionary feature enables desirable sharing. Given the DAC's advantages in usability, we would like to further analyze why DAC is vulnerable to Trojan horse and vulnerable software. In fact, it has been asserted that "This basic principle of discretionary access control contains a fundamental flaw that makes it vulnerable to Trojan horses." [3]. We first show this assertion is inaccurate.

We dissect a DAC system into two components: the *discretionary policy component* and the *enforcement component*. Take the access control system in UNIX-based systems as an example. The policy component consists of the following features: each file has an owner and a number of permission bits controlling which users can read/write/execute the file. The owner of a file can update these permission bits, which is the discretionary feature of DAC. The policy component specifies only which users are authorized, whereas the actual request are generated by processes (subjects) and not users. The enforcement component fills in this gap. In enforcement, each process has an associated user id (the effective user id) that is used to determine this process's privileges, and there are a number of rules that determine how the effective user id is set. In short, the policy part specifies which users can access what resources and the enforcement part tries to determine on which user's behalf the process is running.

In DAC, the effective user id corresponds to our concept of origin. In the default case, the effective user id is inherited from the parent process. The login process will set the effective user id to be the user account corresponding to the logon user.

As a result, the effective user id in DAC is actually the user who starts the process. In other words, DAC identifies the origin of a process to be the user who starts the process.

In DAC, each program is associated with a SETUID bit. When the bit is set, the effective user id of the process running the program will be assigned to be the owner of the program. In that case, DAC identifies the origin of the process to be the program owner.

1.1.2 Defeating Remote Exploits: Mandatory Usable Integrity Protection

DAC is vulnerable to remote exploits because by default DAC identifies the origin of a process to be the user who starts the process. Such an assumption is not true given the software contain vulnerabilities.

Let's consider a typical scenario of the remote exploitation attack. The adversary first exploits a vulnerability (e.g., buffer overflow) in a running server daemon (e.g., a Apache web server) and successfully injects a piece of malicious code. The malicious code spawns a shell, which in turn loads a kernel-mode RootKit by invoking the command insmod. Usually the Apache web server is started by the system administrator and has all system privileges. In this way, the adversary has enough privilege to load the kernel module and successfully takes over the system.

In the above example, the process insmod is started by the administrator, but is controlled by the adversary. A server daemon that has received network traffics may have been exploited and controlled by an attacker and may no longer behave as its design. As a result, when the shell spawned by the server daemon wants to load a kernel module, the real origin behind this request could be the remote adversary if the server is possibly vulnerable.

The analysis above illustrates that, when software contain exploitable vulnerabilities, to determine the real origin of the requests issued by the current process, one has to consider the history information of the current process, the parent process

who created the current process, the process who created the parent process, and so on. For example, if insmod is started by a series of processes that have never communicated with the network, then this means that this request is from a user who logged in through a local terminal. Such a request should be authorized, because it is almost certainly not an attacker, unless an attacker gets physical access to the host, in which case not much security can be provided anyway. On the other hand, if insmod is started by a shell that is a descendant of the HTTP daemon process, then this is almost certainly a result from an attack; the HTTP daemon and its legitimate descendants have no need to load a kernel module.

To capture those history information in identifying the origins of the requests, we propose the Usable Mandatory Integrity Protection (UMIP) model. In the UMIP model, we associate each process with an integrity level, which is either high or low. If a process is likely to be exploited and controlled by an attacker, then the process has low integrity. If a process may be used legitimately for system administration, then the process needs to be high-integrity. By adding the integrity level to the origins of a process, we can make the DAC resistent to the remote exploitation attacks.

The security goal of the UMIP model is to preserve system integrity in the face of network-based attacks. UMIP assumes that programs contain bugs and can be exploited, but the attacker does not have physical access to the host to be protected. UMIP partitions the processes into high-integrity and low-integrity using dynamic information flow and the default policy disallow the low-integrity processes to perform sensitive operations. We discuss the design and implementation details of UMIP in Chapter 3.

1.1.3 Defeating Trojan Horses: Information Flow Enhanced Discretionary Access Control

UMIP is designed to protect the host integrity against the remote adversary. However, UMIP cannot defend against the Trojan horse attacks launched by the

malicious local users because the local users are trusted in UMIP. In order to enhance DAC to be resilient to the Trojan horses, we first consider a typical scenario of the attack.

In a Trojan horse attack, a malicious local user (Bob) plants a Trojan by leaving a malicious executable in the local file system. The malicious executable pretends to be a benign utility program, which is later executed by another local user (Alice). In DAC, the process running the Trojan program will have Alice as the origin. As a result, the Trojan horse will carry Alice's identity and has all the privileges associated with Alice. In this example, when Alice executes a program controlled by Bob, DAC identifies the origin to be Alice (the invoker); this is true only when the program is benign and acts as intended by the invoker. When the program is possibly malicious, both Alice and Bob may affect the requests made by the process. That is, the origin of the process should be a set containing both Alice (the invoker) and Bob (the controller of the program).

Besides planting a Trojan horse, a malicious local user could exploit the vulnerabilities contained in the setuid-root programs to gain root privileges. The same idea above can be applied to defeat local exploits. After reading data, the program may have been exploited by the potentially maliciously formed data. If the program is not assumed to be correct, the controllers of the input data must be added to the set of origins.

This idea is actually an extension of that adds a one-bit integrity-level to the origins in UMIP. In UMIP, to defend against the network adversary, only the network communication is considered as a contamination source to be tracked in identifying the true origins. Similarly, to defend against the malicious local users, we can treat each local user account as a separate contamination source. The origin of a process becomes a set of user accounts that may affect the requests made by the process.

To implement this idea, we propose the Information-Flow Enhanced Discretionary Access Control (IFEDAC) model. IFEDAC keeps the discretionary policy component, but change the enforcement component. In the enforcement component, one should

maintain a set of principals as the origins, rather than a single one, for each process. When a request occurs, it is authorized only when every principal in the set is authorized according to the DAC policy, since any of those principals may be responsible for the request. The origins of a process is tracked using dynamic information flow.

IFEDAC is the first DAC model that can defend against Trojan horses and attacks exploiting buggy software. This is achieved by precisely identifying and fixing what makes DAC vulnerable to Trojan horses and buggy programs. IFEDAC can significantly strengthen end host security, while preserving to a large extent DAC's ease of use. The design and implementation details of IFEDAC is presented in Chapter 4.

1.2 Browser Access Control

The access control in the browser has two goals: (1) to protect the browser and the underlying operating system from being compromised by the adversary (2) to protect the session with one (sensitive) website from being interfered by another (malicious) website. The first goal can be partially addressed by the access control enforced in the operating system. We focus on addressing the second goal.

Modern browsers support tab-browsing. When browsing the Internet, a user typically visits multiple websites simultaneously or sequentially within a single browser session. The websites visited by the user may include both sensitive websites (e.g., online banking, online shopping) and malicious websites. It is critical for the browser to appropriately isolate the browsing sessions with different websites in order to prevent cross-site interference. This is enforced by the same-origin policy (SoP) model [4], which prevents the scripts downloaded from one website from accessing the HTML documents downloaded from another website. However, with today's web architecture, the SoP model implemented in major browsers is not sufficient to ensure the integrity of user sessions. In particular, we consider two types of attacks, the *pretty-bad-proxy adversary against HTTPS* and the *cross-site request forgery* attack.

1.2.1 Pretty-Bad-Proxy Adversary against HTTPS

HTTPS is an end-to-end cryptographic protocol for securing web traffic over insecure networks. Authenticity and confidentiality are the basic promises of HTTPS. When a client communicates with a web server using HTTPS, we expect that: i) no HTTPS payload data can be obtained by a malicious host on the network; ii) the server indeed bears the identity shown in the certificate; and iii) no malicious host in the network can impersonate an authenticated user to access the server. These properties should hold as long as the end systems, i.e. the browser and the server, are trusted.

In other words, the adversary model of HTTPS is simple and clear: the network is completely owned by the adversary, meaning that no network device on the network is assumed trustworthy. The protocol is rigorously designed, implemented and validated using this adversary model. If HTTPS is not robust against this adversary, it is broken by definition.

This work is motivated by our curiosity about whether the same adversary that is carefully considered in the design of HTTPS is also rigorously examined when HTTPS is integrated into the browser. In particular, we focus on an adversary called "Pretty-Bad-Proxy" (PBP), which is a man-in-the-middle attacker that specifically targets the browser's rendering modules above the HTTP/HTTPS layer in order to break the end-to-end security of HTTPS.

With a focused examination of the PBP adversary against various browser behaviors, we realize that PBP is indeed a threat to the effectiveness of HTTPS deployments. We have discovered a set of PBP-exploitable vulnerabilities in IE, Firefox, Opera, Chrome browsers and many websites. By exploiting the vulnerabilities, a PBP can obtain the sensitive data from the HTTPS server. It can also certify malicious web pages and impersonate authenticated users to access the HTTPS server. The existence of the vulnerabilities clearly undermines the end-to-end security guarantees of HTTPS.

The underlying reason that makes the PBP attacks feasible is because the browsers do not correctly identify the origins of web messages in implementing the same-origin policy. To ensure the integrity of an HTTPS session, we need to ensure the integrity of every web message involved in the session. However, with the current deployment of the HTTPS protocol in major browsers, a PBP adversary is able to inject a malicious web message into an HTTPS session and the browser will accept the malicious message as it comes from the target HTTPS domain. These vulnerabilities should be fixed by correctly identifying the origin of every web message in the HTTPS sessions and only accepting those coming from the target HTTPS domain and transmitted over a secure connection. We have reported the vulnerabilities to major browser venders and all vulnerabilities are acknowledged by the vendors. Some of the vulnerabilities have been fixed in the latest versions and other fixes are proposed for next versions. The details about the vulnerabilities we discovered is presented in Chapter 5.

1.2.2 Cross-Site Request Forgery

Cross-site request forgery, also known as one-click attack or session riding and abbreviated as CSRF or XSRF, is an attack against web users [5–7]. In a CSRF attack, a malicious web page instructs a victim user's browser to send a request to a target website. If the victim user is currently logged into the target website, the browser will append authentication tokens such as cookies to the request, authenticating the malicious request as if it is issued by the user. Consequences of CSRF attacks are serious, e.g., an attacker can use CSRF attacks to perform financial transactions with the victim user's account, such as sending a check to the attacker, purchasing a stock, purchasing products and shipping to the attacker. A detailed description of the attack is given in Section 6.2.

The CSRF attack is an attack that leads to the interference of the browsing sessions with different websites. By browsing a malicious webpage, an unintended operation is performed in the sensitive website without the user's knowledge. The

CSRF attack cannot be prevented by the SoP model. The SoP model does not restrict a website to send a request to a different website, because cross-site request is a common feature in the web design (e.g., considering advertisements contained in webpages). However, when the cross-site request leads to sensitive consequence (e.g., financial consequence) the request may be malicious.

Because the sensitive requests typically occur in authenticated sessions, these requests have authentication tokens attached. The reason that makes the CSRF attack feasible is because the browser *always* attaches the user's authentication token to every request. Such a design assumes every request sent by the browser reflects the user's intention. However, the reality is that the browser can be easily tricked into sending a sensitive request that does not reflect the user's intention. When the authentication token is attached to an unintended sensitive request, the CSRF attack happens.

To defeat the CSRF attack, we propose a browser-side solution to enhance the access control in browsers to ensure that *all sensitive requests sent by the browser should reflect the user's intention.* Instead of blindly attaching authentication tokens to every request, the browser infers whether the request reflects the user's intention by considering how the request is triggered and crafted in the browser and which website(s) should be responsible for the request. Sensitive authentication tokens are only attached to the requests that are considered to reflect the user's intention. The design and implementation details of the protection mechanism is discussed in Chapter 6.

1.3 Organization

In the next chapter, we review the related works in operating system access control, HTTPS security and CSRF defenses. We describe the design and implementation of the UMIP model in Chapter 3. The IFEDAC model is described in Chapter 4. In Chapter 5, we present the PBP vulnerabilities we discovered in the major browsers

and the suggested mitigation. We discuss the design and implementation of the browser-side defense against the CSRF attack in Chapter 6. Last, we conclude in Chapter 7.

2 LITERATURE REVIEW

In this chapter, we review the related works in operating system access control and browser access control.

2.1 Related Works in Operating System Access Control

In this section, we review the related works in operating system access control. We first describe the traditional DAC mechanisms implemented in UNIX and Windows. After that, we discuss the research efforts that add mandatory access control (MAC) to operating systems, such as SELinux [8] and AppArmor [9]. Last, we present other works that are related to our works on enhancing DAC.

2.1.1 Unix Access Control

Perhaps the best known DAC mechanism is the DAC system in the Unix family of operating systems (including Linux, Solaris, and the OpenBSD). This DAC mechanism has many intricate details. Because of the space limit, we can cover only the features that are most relevant to its security and usability here. Three most important concepts in this DAC system are users, subjects, and objects. From the computer's point of view, each account is a user. Each user is uniquely identified by a user id. There is a special user called root; it has user id 0. A user can be a member of several groups, which of which contains a set of users has its members. Every group has a numerical id call group id. Each subject is a process; it issues requests to access resources. Many protected resources are modeled as files, which we call objects. Users can configure the DAC system to specify which users are allowed to access the files. For non-file resources, the access policy is generally fixed by the

system. In the descriptions below, we use subjects and processes interchangeably, and objects and files interchangeably. We dissect this DAC into two components: the policy specification component and the enforcement component.

The policy component determines which users are allowed to access what objects. Each object has an owner (which is a user) and an associated group. In addition, each object has 12 permission bits. These bits include: three bits for determining whether the owner of the file can read/write/execute the file, three bits for determining whether users in the associated group can read/write/execute the file, three bits for determining whether all other users can read/write/execute the file, the SUID (set user id) bit, the SGID (set group id) bit, and the sticky bit. They will be discussed later. The file owner, file group and permission bits are part of the metadata of a file, and are stored on the inode of a file. The owner of a file and the root user can update these permission bits, which is the discretionary feature. In most modern UNIX-based systems, only the root can change the owner of a file.

The policy component specifies only which users are authorized, whereas the actual requests are generated by subjects (processes) and not users. The enforcement component fills in this gap; it tries to determine on which users' behalf a process is executing. Each process has several users and groups associated with it. The most important one is the effective user id (euid), which determines the access privileges of the process. The first process in the system has euid 0 (root). When a user logs in, the process's euid is set to the id corresponding to the user. When a process loads a binary through the execve system call, the new euid is unchanged except when the binary file has the SUID bit set, in which case the new euid is the owner of the file.

The SUID bit is needed mainly because of the granularity of access control using files is not fined-grained enough. For example, the password information of all the users is stored in the file /etc/shadow. A user should be allowed to change her password; however, we cannot allow normal users to write the shadow file, as they can then change the passwords of other users as well. To solve this problem, the system has a special utility program (passwd), through which a user can change her

password. The program is owned by the root user and has its SUID bit set. When a user runs the passwd program, the new process has euid root and can change /etc/shadow to update the user's password. Note that this process (initiated by a user) can perform anything the root user is authorized to do. By setting the SUID bit on the passwd program, the system administrator (i.e., the root user) trusts that the program performs only the legitimate operations. That is, it will authenticate a user and change only that user's password in /etc/shadow. Any program with SUID bit set must therefore be carefully written so that they do not contain vulnerabilities to be exploited.

2.1.2 Windows Access Control

In windows [10], the privileges of a process are determined by the access token the process possesses. When a user logs in windows, either locally or remotely, the operating system will create an access token for the user's processes. The token includes a security id (SID) representing the user, and several SIDs representing the user's groups. The token also includes the privileges of the user, and other access control information. When a process attempts to access some resource, the operating system will decide if the access is granted based on the access token of the process.

A problem with this access control mechanism is that the user's processes are given the full privileges when the user is logged in. Given the popularity of windows system in end-user desktops, the users are often the administrators of their system. In order to make it easy to, e.g., install software and change system setting, the common practice is that the users are often logged in as administrator. This practice exposes a common security hole in the windows operating system. If an attacker is able to launch a Trojan horse attack, or exploit some programs of the user, he/she is able to acquire virtually all the privileges. Then the attacker can change the critical part of the operating system and take over the system silently.

In order to provide both the convenience and security to users, Windows Vista features a access control mechanism called User Access Control (UAC) [10]. In UAC, when an administer logs in, the user is granted two access control tokens instead of just one: an administrator access token with full privileges and a standard user access token. When the user performs normal tasks, e.g., browsing web, reading emails, only the standard user access token is involved in access control decisions. When the user wants to perform some administration tasks, e.g., install a program or a driver, Vista will prompt to ask for user's consent and the administrator access token is used afterwards. Therefore the administrator has more control of when the privileges are used, and a Trojan that can not just take over the system silently, while the attempts to compromise the system will be interposed by the user.

Microsoft Vista introduced a security feature called Mandatory Integrity Control (MIC) [10]. The purpose of MIC is to protect critical system objects from attacks and user errors. MIC assigns an integrity level to each object. When a subject is to access an object, the access is granted only when the caller has a higher or equal level as the object. While described in many articles about new security features in Vista before its release, it appears that only a limited form of it is enabled, and only for Internet Explorer, under the name Internet Explorer Protection Mode (IEPM).

2.1.3 Security Enhanced Linux (SELinux)

Security-Enhanced Linux (SELinux) [11] is a security mechanism in Linux that has been developed to support a wide range of security policies especially Mandatory Access Control policies. The development involved several parties including National Security Agency. The mechanism was first implemented as kernel patches and currently it is implemented within the Linux Security Module (LSM) framework. The architecture of SELinux separates policy decision-making logic from the policy enforcement logic. Different security policies can be defined to enforce different high-level security requirements. To date, SELinux policies include features as Type

Enforcement, Role-Based Access Control, Multi-Level Security, etc. We discuss the architecture of SELinux and some policies in this section.

In SELinux, every subject (process) and object (resources like files, sockets, etc) is given a security context, which is a set of security attributes. The objects (resources) are categorized into object security classes. Each object security class represents a specific kind of object, e.g., regular files, folders, TCP sockets, etc. For every object security class, there is a set of access operations that can be performed, e.g., the operations to a file include read/write/execute, lock, create, rename, getattr/setattr, link/unlink, etc. When an access attempt is made by a process, the enforcement part will decide whether to grant this access based on the security contexts of the process, the resource being accessed and the object security class of the resource. The security policy defines whether a process with a particular security context can access an object of a particular security class with a particular security context; if it can, what operations are permitted upon this object. The security policy also defines how the security context changes after some accesses are performed, or how to label the security context of a newly created object.

We use the Type Enforcement (TE) to illustrate how SELinux policy works. In TE every process has a domain and every object (e.g., files) has a type. All processes with a same domain are treated identically, and all objects with a same type are treated identically. The SELinux policy defines what types can be executed by each domain. Also, for each domain the policy defines several types as entrypoint programs, and processes can enter a domain only by executing those entrypoint programs for this domain.

To have fine-grain control of the accesses, there are several types of rules in the policy:

- A TE access vector rule defines the access vector for combination of a domain, a type and an object security class. For example, the following rule

```
allow sshd_t sshd_exec_t:file read execute entrypint;
```

says a process with domain sshd_t domain can read a file with sshd_exec_t type, and a process can enter sshd_t domain by executing a file with sshd_exec_t type. Besides the allow access vector, access vectors can also be defined for auditallow, auditdeny and dontaudit. Operations are denied unless in the policy there is an explicit allow rule. When an operation is granted, it is always not logged unless there is an auditallow. And when an operation is denied, it is logged less there is a dontaudit rule.

- A TE transition rule for a process defines what new domain a process will enter after executing a program, based on the current process domain and the type of the program. For example, the flowing rule

  ```
  type_transition initrc_t sshd_exec_t:process sshd_t;
  ```

  says when a process with domain initrc_t executes a program with the type sshd_exec_t, the process transitions to domain sshd_t.

- A TE transition rule for an object defines the type for a new created object. For example, the following rule

  ```
  type_transition sshd_t tmp_t:dir file shsd_tmp_t;
  ```

  says when a process with domain sshd_t creates a file or a directory in a directory with type tmp_t, the newly created file or directory should be with the type sshd_tmp_t.

SELinux adopts the approach that MAC information is independent from DAC. For example, the users in SELinux are unrelated with the users in DAC, each file needs to be given a label. This requires the file system to support additional labeling, and limits the applicability of the approach. Furthermore, labeling files is a labor-intensive and error-prone process. Each installation of a new software requires update to the policy to assign appropriate labels to the newly added files and possibly

add new domains and types.SELinux policies are difficult to understand by human administrators because of the size of the policy and the many levels of indirection used, e.g., from programs to domains, then to types, and then to files.

2.1.4 AppArmor

AppArmor [9, 12] is an access control system that confines the access permissions on a per program basis. The basic idea is following: for every protected program, AppArmor defines a list of permitted accesses, including file accesses and capabilities. The list for a program is called the program's profile. And the profiles of all protected programs constitute a AppArmor policy. A program's profile contains all possible file reads, file writes, file executions and capabilities that may be performed by a protected program. Under AppArmor, a process that executes a protected program can only perform accesses in the program's profile. By confining program accesses, AppArmor makes local and remote exploits more difficult. Suppose a system is running an FTP server with root account. If an attacker exploits a vulnerability in the server and injects her own code, under normal Linux DAC protection, the attacker is able to gain the full privileges in the system. The attacker can, e.g., install a rootkit by loading a kernel module. However, if the system is protected by AppArmor, there will not be a kernel module loading capability in the FTP server's profile because a FTP server wouldn't need that. Then even if the attacker controls the server process, she cannot directly install a rootkit. Following is an excerpt from a profile for passwd [9]. The profile guarantees that if a local user exploits this setuid root program, the user cannot get full privileges of root.

```
1.  . . .
2. /usr/bin/passwd {
3. . . .
4. capability chown,
5. capability sys_resource,
```

```
6. /etc/.pwd.lock w,
7. /etc/pwdutils/logging r,
8. /etc/shadow rwl,
9. /etc/shadow.old rwl,
10. /etc/shadow.tmp rwl,
11. /usr/bin/passwd mr,
12. /usr/lib/pwdutils/lib*.so* mr,
13. /usr/lib64/pwdutils/lib*.so* mr,
14. /usr/share/cracklib/pw_dict.hwm r,
15. /usr/share/cracklib/pw_dict.pwd r,
16. /usr/share/cracklib/pw_dict.pwi r,
17. }
```

In the profile, there are 2 rules for capabilities (line 4 and 5) and 11 rules for file accesses (line 6 through line 16). A file rule consists of a file name and several permitted access modes. There are totally 9 access modes: read mode, write mode, discrete profile execute mode, discrete profile execute mode - clean exec, unconstrained execute mode, unconstrained execute mode (clean exec), inherit execute mode, Allow PROT_EXEC with mmap(2) calls, link mode. For details of the semantics of the access modes, please refer to [9].

A profile can be created by the program developer and ships with the program. Also, the user can create a profile by AppArmor utilities. A user can run a program in a "learning mode". In this mode, all the permissions of a program is permitted and logged. The user makes the program perform as many accesses as possible. Later the user can use the logs to create the profile of the program. For each access, an AppArmor utility asks the user whether to allow the access; and if the access is a file access, the user can choose to generalize the access by using wildcards in the permitted filename (globbing).

AppArmor also provides finer-grain access control than process level, by the "ChangeHat" feature. ChangeHat-aware programs can use this feature to have part of a program using a different profile. For more details please refer to [9].

The approach in AppArmor identifies a number of programs that, when compromised, could be dangerous, and confine them by a policy. If a program has no policy associated with it, then it is by default not confined, and if a program has a policy, then it can access only the objects specified in the policy. This approach remains vulnerable to Trojan horse attacks. As most programs, such as shells, obtained through normal usage channels are unconfined, the execution of a trojan horse program will not be subject to the control of the system.

Regarding policy design, AppArmor uses the same approach as the Targeted Policy in Fedora Core Linux, i.e., if a program has no policy associated with it, then it is by default not confined, and if a program has a policy, then it can access only the objects specified in the policy. This approach violates the fail-safe defaults principle [13], as a program with no policy will by default run unconfined. AppArmor does not maintain integrity levels for processes or files, and thus cannot differentiate whether a process or a file is contaminated or not. For example, without tracking contamination, one cannot specify a policy that system administration through X clients are allowed as long as the X server and other X clients have not communicated with the network. Also, AppArmor cannot protect users from accidentally downloading and executing malicious programs.

2.1.5 Other Related Works in Operating System Access Control

The limitations of DAC have been discussed in many sources, e.g., [3, 14]. Traditionally, people deal with the weaknesses of DAC by replacing or enhancing it with Mandatory Access Control (MAC). There are three classes of approaches to add MAC to operating systems: confidentiality-based, confinement-based, and integrity-based.

Perhaps the best known example of confidentiality-based MAC is the Bell La-Padula (BLP) model [15]. Systems that implement protection models similar to BLP include Trusted Solaris and IX [16]. The BLP model assumes that programs are either trusted or untrusted. This results in a strict security policy rule (the *-property) that will break today's COTS operating systems, unless almost all components are declared to be trusted.

Confinement-based MAC systems include SELinux, systrace [17], securelevel [18] and LIDS [19], while flexible and powerful, require extensive expertise to configure. These systems focus on mechanisms, whereas our approach focuses on providing a policy model that achieves a high degree of protection without getting in the way of normal operations. Systrace [17] defines policies for programs at a finer granularity. Instead of defining allowed accesses to files and capabilities, systrace defines allowed system calls with parameter values for each program to confine the operations of processes. PACL [20] also uses the idea of limiting the programs that can access certain objects. It uses an access control list for each file to store the list of programs that are allowed to access the file. Securelevel [18] is a security mechanism in *BSD kernels. When the securelevel is positive, the kernel restricts certain tasks; not even the superuser (i.e., root) is allowed to do them. Any superuser process can raise securelevel, but only the init process can lower it. The weakness of securelevel is clearly explained in the FreeBSD FAQ [18]: "*One of its biggest problems is that in order for it to be at all effective, all files used in the boot process up until the securelevel is set must be protected. If an attacker can get the system to execute their code prior to the securelevel being set [...], its protections are invalidated. While this task of protecting all files used in the boot process is not technically impossible, if it is achieved, system maintenance will become a nightmare since one would have to take the system down, at least to single-user mode, to modify a configuration file.*"

These systems require the specification of a new access matrix (typically with programs as one axis and files as another axis) separately from the existing DAC mechanism. While being flexible, it is often overwhelming for end users to configure

them. We aim at achieving the protection objective in the DAC mechanism and thus reuse the DAC information.

Another approach to introduce MAC to operating systems is to protect the integrity of critical system objects. The Biba model [21] is perhaps the earliest mandatory integrity protection model. In the model each subject and each object has an integrity level. Biba defines five policies for permission checking and label updating. For example, one policy is the strict integrity policy, in which subject and object integrity labels never change, and a subject can read only objects with a higher (or equal) integrity level and can write only objects whose integrity level with a lower (or equal) integrity level. LOMAC [22] is an implementation in operating systems of a policy from Biba called subject low-water mark policy. Each object is assigned an integrity level. Once assigned, an object's level never changes. A subject's integrity level drops when it reads a low-integrity objects. It aims at protecting system integrity and places emphasis on usability.

The approaches in AppArmor, systrace, and PACL are to identify a number of programs that, when compromised, could be dangerous, and confine them by a policy. These techniques require a large policy, because they do not have default policy rules to allow some accesses and must explicitly specify every access. Furthermore, these approaches remain vulnerable to trojan horse attacks. As most programs, such as shells, obtained through normal usage channels are unconfined, the execution of a trojan horse program will not be subject to the control of the system.

Another well-known integrity model is the Clark-Wilson model [23], with follow-up work by Karger [24] and Lee [25], among others. These integrity-protection approaches have not been applied to operating systems and do not support user-specific integrity, e.g., separating one user from another.

McCollum et al. [26] discussed new forms of access control other than MAC and DAC, which combines the information flow mechanism used in MAC and the attributes and identity based privileges in DAC. The approach in [26] assigns attributes labels and owner information to both objects and subjects. The labels propagate

based on information flow. [26] was designed to protect confidentiality and aimed at the special requirements of the DoD/intelligence for automated information analysis. This work introducing new kinds of policy beyond DAC and MAC, whereas our work applies MAC techniques to strengthen DAC in operating systems,

Hicks et al. [27] proposed an architecture for an operating system service that integrates a security-typed language with MAC in operating systems, and built SIESTA, an implementation of the service that handles applications developed in Jif running on SELinux.

Dynamic information flow tracking within programs has been used to detect attacks and generate signatures of attacks. Examples include TaintCheck [28] and taint-enhanced policy enforcement [29]. These techniques provide protections orthogonal to ours. They defend particular programs against being exploited by attackers. We focus on general operating system access control techniques that apply to all processes. The idea that a request may represent the intention of one of many principals appeared also in the work of Adabi et al. on access control in distributed systems [30]. There are several recent works on developing new operating system access control models that use information flow. Asbestos [31], HiStar [32], and Fluke [33] use decentralized information flow control, which allows application writers to control how data flows between pieces of an application and the outside world. We have a different goal of fixing DAC without affecting application programmers.

2.2 Related Works in Browser Security and HTTPS Security

In this section, we review the related works in browser security in general, with a focus on HTTPS security.

Violations of the same-origin policy are one of the most significant classes of security vulnerabilities on the web. Classic examples include cross-site scripting (aka, XSS) and browser's domain-isolation bugs: (1) XSS is commonly considered as a web application bug. Vulnerable web applications fail to perform sanity checks for user

input data, but erroneously interpret the data as scripts in the web application's own security. Many researchers have proposed techniques to address XSS bugs. A compiler technique is proposed by Livshits and Lam to find XSS bugs in Java applications [34]. Based on the observation that XSS attacks require user-input data be executed at runtime, Xu et al proposed using taint tracking to detect the attacks [35]. There are many other research efforts in the area of XSS that we cannot cite due to space constraints. (2) Historically all browser products had bugs in their domain-isolation mechanisms, which allow a frame tagged as evil.com to access the document in another frame tagged as victim.com on the browsers. Security vulnerability databases, including SecurityFocus.com, have posted many bugs against IE, Firefox, Opera, etc. These vulnerabilities are discussed in [36].

People already understand that HTTPS security is contingent upon the security of clients, servers and certificate authorities. Binary-executable-level threats, such as buffer overruns, virus infections and incautious installations of unsigned software from the Internet, allow malicious binary code to jump out of the browser sandbox. In particular, when a malicious proxy or router is on the communication path, the binary-level vulnerabilities can be exploited even when the browser visits legitimate websites. Unsurprisingly, once the browser's executable is contaminated, HTTPS becomes ineffective. In addition to the binary-level vulnerabilities, XSS bugs and browser's domain-isolation failures may compromise HTTPS. Furthermore, some certificate authorities use questionable practices in certificate issuance, undermining the effectiveness of HTTPS. For example, despite the know weaknesses of MD5, some certificate authorities have not completely discontinued the issuance of MD5-based certificated. Sotirov, Stevens, et al have recently shown the practical threat of the MD5 collision by creating a rogue certificate authority certificate [37]. In contrast to these known weaknesses, the contribution of our work is to emphasize that the high-level browser modules, such as the HTML engine and the scripting engine, is not thoroughly examined against the PBP adversary, and PBP indeed has its uniqueness in attacking the end-to-end security.

HTTPS has usability problems because of its unfriendliness to average users. Usability studies have shown that most average users do not check the lock icon when they access HTTPS websites [38]. They are willing to ignore any security warning dialog, including the warning of certificate errors. Logic bugs in browsers' GUI implementations can also affect HTTPS effectiveness. In [39], we show a number of logic bugs that allow an arbitrary page to appear with a spoofed address and an SSL certificate on the address bar.

The HPIHSL vulnerability described in Section 5.3.3 is related to the "mixed content" vulnerability in [40] by Jackson and Barth. Jackson/Barth and we exchanged the early drafts of [40] and this paper in October 2007 to understand the findings made by both parties, which are distinguishable in the following aspects: (1) the scenario in [40] is that the developer of an HTTPS page accidentally embeds a script, an SWF movie or a Java applet using HTTP, while our main perspective is about loading an HTTP-intended page through HTTPS; (2) we discover that the warning message about an HTTP script in an HTTPS frame can be suppressed by placing the HTTPS frame in a HTTP top-level window, while [40] argues that such a warning is often ignored by users; (3) we found twelve concrete e-commerce and e-service sites that we sampled where the vulnerability based on HPIHSL pages exists. This suggests that this vulnerability may currently be pervasive. In [40], there is no argument about the pervasiveness of the accidental HTTP-embedding mistakes made by developers.

Karlof, Shankar, et al envision an attack called "dynamic pharming" to attack HTTPS sessions by a third-party website. The attack is based on the assumption that the victim user accepts a faked certificate [41]. Because HTTPS security crucially relies on valid certificates, accepting a faked certificate is a sufficient condition to void HTTPS guarantees. To address dynamic pharming, the authors propose locked same-origin-policies to enhance the current same-origin-policy. These policies do not cover PBP attacks discussed in Sections III.B, III.C, IV.A and IV.B. For the attack in Section III.A, if developers understand that 4xx/5xx pages from the proxy cannot bear the context of the target server, then the current same-origin-policy is already

secure; if they overlook this, as all browser vendors did, it is unlikely that the mistake can be avoided in the implementations of the locked same-origin-policies.

Researchers have found vulnerabilities in DNS and WPAD protocol implementations. Kaminsky showed the practicality of the DNS cache poisoning attack, which can effectively redirect the victim machine's traffic to an IP address specified by the attacker [42]. This attack can be used to fool the user to connect to a malicious proxy. Researchers also found security issues about WPAD, e.g., registering a hostname "wpad" in various levels of the DNS hierarchies can result in the retrievals of PAC scripts from problematic or insecure PAC servers [43, 44]. Unlike these findings, our work does not attempt to show any vulnerability in WPAD. It is unsurprising that the communication over an unencrypted channel is insecure when the attacker can sniff and send packets on the network - several possibilities of maliciously configuring the browser's proxy settings were documented in [45]. We discuss PAC, WPAD and the manual proxy setting only as a feasibility argument of the PBP vulnerabilities.

2.3 Existing CSRF Defenses

Several defense mechanisms have been proposed for CSFR attacks, we now discuss their limitations.

Filtering authentication tokens from cross-site requests. Johns et al. [46] proposed a client-side proxy solution, which stripes all authentication tokens from a cross-site request. The proxy intercepts web pages before they reach the browser and appends a secret random value to all URLs in the web page. Then the proxy removes the authentication tokens from the requests that do not have a correct random value. The solution breaks the auto-login feature and content sharing websites (such as Digg, Facebook, etc.) because it does not distinguish legitimate cross-site requests from malicious cross-site requests. In addition, it does not support HTML dynamically created in the browser and cannot work with SSL connections.

Authenticating web forms. The most popular CSRF defense is to authenticate the web form from which an HTTP request is generated. This is achieved by having a shared random secret, called a as a *secret validation token*, between the web form and the web server. If a web form provides a sensitive service, the web server embeds a secret validation token in an invisible field or the POST action URL of the form. Whenever form data is submitted, the request is processed only if it contains the correct secret value. Not knowing the secret, the adversary cannot forge a valid request. One drawback of this approach is it requires nontrivial changes to the web applications. Moreover, as pointed out by Barth et al. [47], although there exist several variants of this technique they are generally complicated to implement correctly. Many frameworks accidentally leak the secret token to other websites. For example, NoForge proposed in [48] leaks the token to other websites through the URL and the HTTP Referer header.

Referer-checking. In many cases, when the browser issues an HTTP request, it includes a Referer header that indicates which URL initiated the request. A web application can defend itself against CSRF attacks by rejecting the sensitive requests with a Referer of a different website. A major limitation with this approach is that some requests do not have a Referer header. There does not exist a standard specification on when to and when not to send the Referer header. Different browser vendors behave differently. Johns and Winter [46] give a summary on when browsers do not send the Referer header in major browsers. As a result, both a legitimate request and a malicious request may lack the Referer header. The adversary can easily construct a request lacking the Referer header. Moreover, because the Referer header may contain sensitive information that impinges on the privacy of web users, some users prohibit their browsers to send Referer header and some network proxies and routers suppress the Referer headers. As a result, simply rejecting the requests lacking a Referer header incurs a compatibility penalty. Barth et al. [47] suggested a new Origin header that includes only the hostname part of the Referer header, to alleviate the privacy concern.

It remains to be seen whether this will be adopted. In conclusion, using a server-side referer-checking to defeat the CSRF attacks has a dilemma in handling the requests that lack a Referer header.

3 USABLE MANDATORY INTEGRITY PROTECTION

3.1 Motivation

Host compromise is one of the most serious computer security problems today. Two key reasons why hosts can be easily compromised are: (1) software are buggy, and (2) the discretionary access control mechanism in today's operating systems is insufficient for defending against network-based attacks. A traditional approach to address DAC's weaknesses is to add mandatory access control (MAC) to the existing DAC in operating systems. There are a lot of research efforts on making computer systems more secure by adding MAC[1] to operating systems, e.g., Janus [49], DTE Unix [50, 51], Linux Intrusion Detection System (LIDS) [19], LOMAC [22], systrace [17], AppArmor [9, 12], and Security Enhanced Linux (SELinux) [8]. Several of these systems are flexible and powerful. Through proper configuration, they could result in highly-secure systems. However, they are also complex and intimidating to configure.

For example, SELinux has 29 different classes of objects, hundreds of possible operations, and thousands of policy rules for a typical system. The SELinux policy interface is daunting even for security experts. While SELinux makes sense in a setting where the systems run similar applications, and sophisticated security expertise is available, its applicability to a more general setting is unclear.

In this chapter, we tackle the problem of designing and implementing a usable MAC system to protect end hosts. We start by identifying several principles for designing usable access control mechanisms in general. We then introduce the Usable

[1] In this dissertation, we use MAC to refer to the approach where a system-wide security policy restricts the access rights of processes. This is a wider interpretation of MAC than that in the TCSEC [2], which focuses on multi-level security.

Mandatory Integrity Protection (UMIP) model, which was designed following these principles.

3.2 Design Principles for Usable Access Control Systems

While it is widely agreed that usability is very important for security technologies, how to design an access control system that has a high level of usability has not been explored much in the literature. In this section we present six principles for designing usable access control systems. Some of these principles challenge established common wisdom in the field, because we place an unusually high premium on usability. These principles will be illustrated by our design of UMIP in Section 3.

Principle 1 *Provide "good enough" security with a high level of usability, rather than "better" security with a low level of usability.*

Our philosophy is that rather than providing a protection system that can theoretically provide very strong security guarantees but requires huge effort and expertise to configure correctly, we aim at providing a system that is easy to configure and that can greatly increase the level of security by reducing the attack surfaces. Sandhu [52] made a case for good-enough security, observing that "*cumbersome technology will be deployed and operated incorrectly and insecurely, or perhaps not at all.*" Sandhu also identified three principles that guide information security, the second of which is "*Good enough always beats perfect*"[2]. He observed that the applicability of this principle to the computer security field is further amplified because there is no such thing as "perfect" in security, and restate the principle as "*Good enough always beats 'better but imperfect'.*"

There may be situations that one would want stronger security guarantees, even though the cost of administration is much more expensive. However, to defend against threats such as botnets, one needs to protect the most vulnerable computers on the

[2]The first one is "*Good enough is good enough*" and the third one is "*The really hard part is determining what is good enough.*"

Internet, i.e., computers that are managed by users with little expertise in system security. One thus needs a protection system with a high level of usability.

One corollary following from this principle is that *sometimes one needs to tradeoff security for simplicity of the design.* Below we discuss five other principles, which further help achieve the goal of usable access control.

Principle 2 *Provide policy, not just mechanism.*

Raymond discussed in his book [53] the topic of "*what UNIX gets wrong*" in terms of philosophy, and wrote "*perhaps the most enduring objections to Unix are consequences of a feature of its philosophy first made explicit by the designers of the X windowing system. X strives to provide 'mechanism, not policy'. [...] But the cost of the mechanism-not-policy approach is that when the user can set policy, the user must set policy. Nontechnical end-users frequently find Unix's profusion of options and interface styles overwhelming.*"

The mechanism-not-policy approach is especially problematic for security. A security mechanism that is very flexible and can be extensively configured is not just overwhelming for end users, it is also highly error-prone. While there are right ways to configure the mechanism to enforce some desirable security policies, there are often many more incorrect ways to configure a system. And the complexity often overwhelms users so that the mechanism is simply not enabled.

This mechanism-not-policy philosophy is implicitly used in the design of many MAC systems for operating systems. For example, systems such LIDS, systrace, and SELinux all aim at providing a mechanism that can be used to implement a wide range of policies. While a mechanism is absolutely necessary for implementing a protection system, having only a low-level mechanism is not enough.

Principle 3 *Have a well-defined security objective.*

The first step of designing a policy is to identify a security objective, because only then can one make meaningful tradeoffs between security and usability. To make

tradeoffs, one must ask and answer the question: if the policy model is simplified in this way, can we still achieve the security objective? A security objective should identify two things: what kind of adversaries the system is designed to protect against, i.e., what abilities does one assume the adversaries have, and what security properties one wants to achieve even in the presence of such adversaries. Often times, MAC systems do not clearly identify the security objective. For example, achieving multi-level security is often identified together with defending against network attacks. They are very different kinds of security objectives. History has taught us that designing usable multi-level secure systems is extremely difficult, and it seems unlikely that one can build a usable access control system that can achieve both objectives.

Principle 4 *Carefully design ways to support exceptions in the policy model.*

Given the complexity of modern operating systems and the diverse scenarios in which computers are used, no simple policy model can capture all accesses that need to be allowed, and, at the same time, forbid all illegal accesses. It is thus necessary to have ways to specify exceptions in the policy model. The challenges lie in designing the policy model and the exception mechanisms so that the number of exceptions is small, the exceptions are easy and intuitive to specify, the exceptions provide the desired flexibility, and the attack surface exposed by the exceptions is limited. Little research has focused on studying how to support exceptions in an MAC model. As we will see, much effort in designing UMIP goes to designing mechanisms to support exceptions.

Principle 5 *Rather than trying to achieve "strict least privilege", aim for "good-enough least privilege".*

It is widely recognized that one problem with existing DAC mechanisms is that it does not support the least privilege principle [13]. For example, in traditional UNIX access control, many operations can be performed only by the root user. If a program needs to perform any of these operations, it needs to be given the root privilege.

As a result, an attacker can exploit vulnerabilities in the program and abuse these privileges. Many propose to remedy the problem by using very-fine-grained access control and to achieve strict least privilege. For example, the guiding principles for designing policies for systems such as SELinux, systrace, and AppArmor is to identify all objects a program needs to access when it is not under attack and grants access only to those objects. This approach results in a large number of policy rules. We believe that it is sufficient to restrict privileges just enough to achieve the security objective; and this enables one to design more usable access control systems. This principle can be viewed as a corollary of Principle 1. We state it as a separate principle because of the popularity of the least privilege principle.

Principle 6 *Use familiar abstractions in policy specification interface.*

Psychological acceptability is one of the eight principles for designing security mechanisms identified by Salzer and Schroeder [13]. They wrote "*It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. Also, to the extent that the user's mental image of his protection goals matches the mechanisms he must use, mistakes will be minimized. If he must translate his image of his protection needs into a radically different specification language, he will make errors.*" This entails that the policy specification interface should use concepts and abstractions that administrators are familiar with. This principle is violated by systems such as systrace and SELinux.

3.3 The UMIP Model

While the description of the UMIP model in this section is based on our design for Linux, we believe that the model can be applied to other UNIX variants with minor changes. While some (but not all) ideas would be applicable also to non-Unix operating systems such as the Microsoft Windows family, investigating the suitability of UMIP or a similar model for Microsoft Windows is beyond the scope of this thesis work.

We now identify the security objective of our policy model. We aim at protecting the system integrity against network-based attacks. We assume that network server and client programs contain bugs and can be exploited if the attacker is able to feed input to them. We assume that users may make careless mistakes in their actions, e.g., downloading a malicious program from the Internet and running it. However, we assume that the attacker does not have physical access to the host to be protected. Our policy model aims at ensuring that under most attack channels, the attacker can only get limited privileges and cannot compromise the system integrity. For example, if a host runs privileged network-facing programs that contain vulnerabilities, the host will not be completely taken over by an attacker as a bot. The attacker may be able to exploit bugs in these programs to run some code on the host. However, the attacker cannot install rootkits. Furthermore, if the host reboots, the attacker does not control the host anymore. Similarly, if a network client program is exploited, the damage is limited. We also aim at protecting against indirect attacks, where the attacker creates malicious programs to wait for users to execute them, or creates/changes files to exploit vulnerabilities in programs that later read these files.

The usability goals for UMIP are twofold: First, configuring a UMIP system should not be more difficult than installing and configuring an operating system. Second, existing applications and common usage practices can still be used under UMIP. Depending on the needs of a system, the administrator of the system should be able to configure the system in a less-secure, but easier-to-user manner.

One constraint that we have for UMIP is that it can be implemented using an existing mechanism (namely the Linux Security Modules framework).

3.3.1 An Overview of the UMIP Model

An important design question for any operating system access control system is: What is a principal? That is, when a process requests to perform certain operations, what information about the process should be used in deciding whether the request

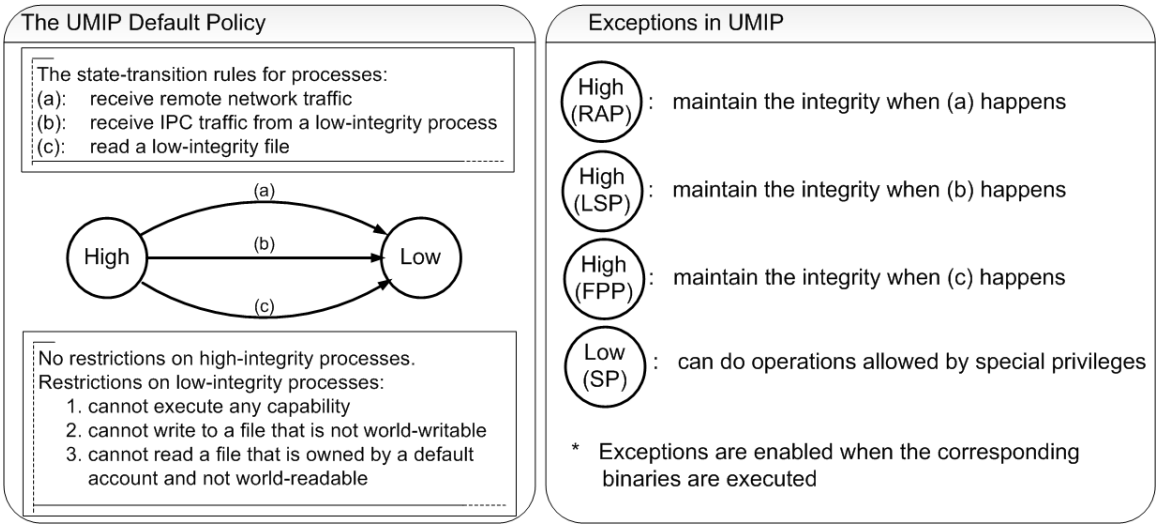

Figure 3.1. The summary of the UMIP model

should be authorized. The traditional UNIX access control system treats a pair of (uid,gid) as a principal. The effective uid and gid together determine the privileges of a process. As many operations can be performed only when the effective uid is 0, many programs owned by the root user are designated setuid. One problem with this approach is that it does not consider the possibility that these programs may be buggy. If all privileged programs are written correctly, then this approach is fine. However, when privileged programs contain bugs, they can be exploited so that attackers can use the privileges to damage the system.

As having just uid and gid is too coarse-granulated, a natural extension is to treat a triple of uid, gid, and the current program that is running in the process as a principal. The thinking is that, if one can identify all possible operations a privileged program would do and only allows it to do those, then the damage of an attacker taking over the program is limited. This design is also insufficient, however. Consider a request to load a kernel module[3] that comes from a process running the program insmod with effective user-id 0. As loading a kernel module is what insmod is supposed

[3]A loadable kernel module is a piece of code that can be loaded into and unloaded from kernel upon demand. LKMs (Loadable Kernel Modules) are a feature of the Linux kernel, sometimes used to add

to do, such access must be allowed. However, this process might be started by an attacker who has compromised a daemon process running as root and obtained a root shell as the result of the exploits. If the request is authorized, then this may enable the installation of a kernel rootkit, and lead to complete system compromise. One may try to prevent this by preventing the daemon program from running certain programs (such as shell); however, certain daemons have legitimate need to run shells or other programs that can lead to running insmod. In this case, a daemon can legitimately run a shell, the shell can legitimately run insmod, and insmod can legitimately load kernel modules. If one looks at only the current program together with (uid,gid), then any individual access needs to be allowed; however, the combination of them clearly needs to be stopped.

The analysis above illustrates that, to determine what the current process should be allowed to do, one has to consider the parent process who created the current process, the process who created the parent process, and so on. We call this the *request channel*. For example, if insmod is started by a series of processes that have never communicated with the network, then this means that this request is from a user who logged in through a local terminal. Such a request should be authorized, because it is almost certainly not an attacker, unless an attacker gets physical access to the host, in which case not much security can be provided anyway. On the other hand, if insmod is started by a shell that is a descendant of the ftp daemon process, then this is almost certainly a result from an attack; the ftp daemon and its legitimate descendants have no need to load a kernel module.

The key challenge is how to capture the information in a request channel in a succinct way. The domain-type enforcement approach used in SELinux and DTE Unix can be viewed as summarizing the request channel in the form of a domain. Whenever a channel represents a different set of privileges from other channels, a new domain is needed. This requires a large number of domains to be introduced.

---

support for new hardware or otherwise insert code into the kernel to support new features. Using LKMs is one popular method for implementing kernel-mode rootkits on Linux.

The approach we take is to use a few fields associated with a process to record necessary information about the request channel. The most important field is one bit to classify the request channel into high integrity or low integrity. If a request channel is likely to be exploited by an attacker, then the process has low integrity. If a request channel may be used legitimately for system administration, then the process needs to be high-integrity. Note that a request channel may be both legitimately used for system administration and potentially exploitable. In this case, administrators must explicitly set the policy to allow such channels for system administration. The model tries to minimize the attack surface exposed by such policy setting when possible.

When a process is marked as low-integrity, this means that it is potentially contaminated. We do not try to identify whether a process is actually attacked. The success of our approach depends on the observation that with such an apparently crude distinction of low-integrity and high-integrity processes, only a few low-integrity processes need to perform a small number of security critical operations, which can be specified using a few simple policies as exceptions.

> **Basic UMIP Model:** Each process has one bit that denotes its integrity level. When a process is created, it inherits the integrity level of the parent process. When a process performs an operation that makes it potentially contaminated, it drops its integrity. A low-integrity process by default cannot perform sensitive operations.

The basic UMIP model is then extended with exceptions to support existing softwares and system usage practices. Figure 3.1 gives an overview of UMIP. A high-integrity process may drop its integrity to low in one of three ways. There are two classes of exceptions that can be specified for programs. The first class allows a program binary to be identified as one or more of: RAP (Remote Administration Point), LSP (Local Service Point), and FPP (File Processing Program). Such exceptions allow a process running the binary to maintain its integrity level when certain events that normally would drop the process's integrity occur. In the second class,

a program binary can be given special privileges (e.g., using some capabilities, reading/writing certain protected files) so that a process running the program can have these privileges even in low integrity.

In the rest of this section, we describe the UMIP model in detail. Section 3.3.2 discusses contamination through network and interprocess communications. Section 3.3.3 discusses restrictions on low-integrity processes. Section 3.3.4 discusses contamination through files. Section 3.3.5 discusses protecting files owned by non-system accounts. Comparison of UMIP with closely related integrity models is given in Section 3.3.6.

3.3.2 Dealing with Communications

When a process receives remote network traffic (network traffic that is not from the localhost loopback), its integrity level should drop, as the program may contain vulnerabilities and the traffic may be sent by an attacker to exploit such vulnerabilities. Under this default policy, system maintenance tasks (e.g., installing new softwares, updating system files, and changing configuration files) can be performed only through a local terminal. Users can log in remotely, but cannot perform these sensitive tasks. While this offers a high degree of security, it may be too restrictive in many systems, e.g., in a collocated server hosting scenario.

In the UMIP model, a program may be identified as a *remote administration point (RAP)*. The effect is that a process running the program maintains its integrity level when receiving network traffic. If one wants to allow remote system administration through, e.g., the secure shell daemon, then one can identify /usr/sbin/sshd as a remote administration point. (Note that if a process descending from sshd runs a program other than sshd and receives network traffic, its integrity level drops.) Introducing RAP is the result of trading off security in favor of usability. Allowing remote administration certainly makes the system less secure. If remote administration through sshd is allowed, and the attacker can successfully exploit bugs in sshd, then the attacker

can take over the system, as this is specified as a legitimate remote administration channel. However, note that in this case the attack surface is greatly reduced from all daemon programs, to only sshd. Some daemon programs (such as httpd) are much more complicated than sshd and are likely to contain more bugs. Moreover, firewalls can be used to limit the network addresses from which one can connect to a machine via sshd; whereas one often has to open the httpd server to the world. Finally, techniques such as privilege separation [54, 55] can be used to further mitigate attacks against sshd. The UMIP model leaves the decision of whether to allow remote administration through channels such as sshd to the system administrators.

We also need to consider what happens when a process receives Inter-Process Communications (IPC) from another local process. UMIP considers integrity contamination through those IPC channels that can be used to send free-formed data, because such data can be crafted to exploit bugs in the receiving process. Under Linux, such channels include UNIX domain socket, pipe, fifo, message queue, shared memory, and shared file in the tmpfs filesystem. In addition, UMIP treats local loopback network communication as a form of IPC. When a process reads from one of these IPC channels which have been written by a low-integrity process, then the integrity level of the process drops, even when the process is a RAP.

Similar to the concept of RAP, a program may be identified as a Local Service Point (LSP), which enables a process running the program to maintain its integrity level after receiving IPC communications from low-integrity processes. For example, if one wants to enable system administration and networking activities (such as web browsing) to happen in one X Window environment, the X server and the desktop manager can be declared as LSPs. When some X clients communicate with network and drop to low-integrity, the X server, the desktop manager and other X clients can still maintain high integrity.

3.3.3 Restricting Low-Integrity Processes

Our approach requires the identification of security-critical operations that would affect system integrity so that our protection system can prevent low-integrity processes from carrying them out. We classify security-critical operations into two categories, file operations and operations that are not associated with specific files.

Examples of non-file administrative operations include loading a kernel module, administration of IP firewall, modification of routing table, network interface configuration, rebooting the machine, ptrace other processes, mounting and unmounting file systems, and so on. These operations are essential for maintaining system integrity and availability, and are often used by malicious code. In modern Linux, these operations are controlled by capabilities, which were introduced since version 2.1 of the Linux kernel. Capabilities break the privileges normally reserved for root down to smaller pieces. As of Linux Kernel 2.6.11, Linux has 31 different capabilities. The default UMIP rule grants only two capabilities CAP_SETGID and CAP_SETUID to low-integrity processes; furthermore, low-integrity processes are restricted in that they can use setuid and setgid only in the following two ways: (1) swapping among effective, real, and saved uids and gids, and (2) going from the root account to another system account. (A system account, with the exception of root, does not correspond to an actual human user.) We allow low-integrity processes to use setuid and setgid this way because many daemon programs do them and they do not compromise our security objective. Note that by this design, a low-integrity process running as root cannot set its uid to a new normal user.

It is much more challenging to identify which files should be considered sensitive, as a large number of objects in an operating system are modeled as files. Different hosts may have different softwares installed, and have different sensitive files. The list of files that need to be protected is quite long, e.g., system programs and libraries, system configuration files, service program configuration files, system log files, kernel image files, and images of the memory (such as /dev/kmem and /dev/mem). We

cannot ask the end users to label files, as our goal is to have the system configurable by ordinary system administrators who are not security experts. Our novel approach here is to utilize the valuable information in existing Discretionary Access Control (DAC) mechanisms.

Using DAC info for MAC All commercial operating systems have built-in DAC mechanisms. For example, UNIX and UNIX variants use the permission bits to support DAC. While DAC by itself is insufficient for stopping network-based attacks, DAC access control information is nonetheless very important. For example, when one installs Linux from a distribution, files such as /etc/passwd and /etc/shadow would be writable only by root. This indicates that writing to these files is security critical. Similarly, files such as /etc/shadow would be readable only by root, indicating that reading them is security critical. Such DAC information has been used by millions of users and examined for decades. Our approach utilizes this information, rather than asking the end users to label all files, which is a labor intensive and error-prone process. UMIP offers both read and write protection for files owned by system accounts. A low-integrity process (even if having effective uid 0) is forbidden from reading a file that is owned by a system account and is not readable by world; such a file is said to be *read-protected*. A low-integrity process is also forbidden from writing to a file owned by a system account and is not writable by world. Such a file is said to be *write-protected*. Finally, a low-integrity process is forbidden from changing the DAC permission of any (read- or write-) protected file.

Exception policies: least privilege for sensitive operations Some network-facing daemons need to access resources that are protected. Because these processes receive network communications, they will be low-integrity, and the default policy will stop such access. We deal with this by allowing the specification of policy exceptions for system binaries. For example, one policy we use is that the binary "/usr/sbin/vsftpd" is allowed to use the capabilities CAP_NET_BIND_SERVICE, CAP_SYS_SETUID, CAP_SYS_SETGID, and CAP_SYS_CHROOT, to read the file /etc/shadow, to read all files under the directory /etc/vsftpd, and to read or write

the file /var/log/xferlog. This daemon program needs to read /etc/shadow to authenticate remote users. If an attacker can exploit a vulnerability in vsftpd and inject code into the address space of vsftpd, this code can read /etc/shadow file. However, if the attacker injects shell code to obtain an shell by exploiting the vulnerabilities, then the exception policy for the shell process will be reset to NULL and the attacker loses the ability to read /etc/passwd. Furthermore, the attacker cannot write to any system binary or install rootkits. Under this policy, an administrator cannot directly upload files to replace system binaries. However, the administrator can upload files to another directory and login through a remote administration channel (e.g., through sshd) and then replace system binary files with the uploaded files.

When a high integrity process loads a program that has an exception policy, the process has special privileges as specified by the policy. Even when the process later receives network traffic and drops integrity, the special privileges remain for the process. However, when a low integrity process loads a program that has an exception policy, the process is denied the special privileges in the policy. The rationale is as follows. Some network administration tools (such as iptables) must perform network communications and will thus drop its integrity, so they need to be given capability exceptions for CAP_NET_ADMIN. However, we would not want a low-integrity process to invoke them and still have the special privileges. On the other hand, some programs need to invoke other programs when its integrity is low, and the invoked program needs special privilege. For example, sendmail needs to invoke procmail when its integrity is low, and procmail needs to write to the spool directory which is write-protected. We resolve this by defining executing relationships between programs. If there is an executing relationship between the program X to the program Y, then when a process running X executes Y, even if the process is in the state of low-integrity, the process will have the special permissions associated with Y after executing. In the example, we define an executing relationship from sendmail to procmail and give procmail the special permission to write to the spool directory.

3.3.4 Contamination through Files

As an attacker may be able to control contents in files that are not write-protected, a process's integrity level needs to drop after reading and executing files that are not write-protected. However, even if a file is write-protected, it may still be written by low-integrity processes, due to the existence of exception policies. We use one permission bit to track whether a file has been written by a low-integrity process. There are 12 permission bits for each file in a UNIX file system: 9 of them indicate read/write/execute permissions for user/group/world; the other three are setuid, setgid, and the sticky bit. The sticky bit is no longer used for regular files (it is still useful for directories), and we use it to track contamination for files. When a low-integrity process writes to a file that is write-protected as allowed by an exception, the file's sticky bit is set. A file is considered to be low-integrity (potentially contaminated) when either it is not write-protected, or has the sticky bit set.

When a process reads a low-integrity file, the process's integrity level drops. We do not consider reading a directory that was changed by a low-integrity process as contamination, as the directory is maintained by the file system, which should handle directory contents properly. When a file's permission is changed from world-writable to not world-writable, the sticky bit is set, as the file may have been contaminated while it was world-writable.

A low-integrity process is forbidden from changing the sticky bit of a file. Only a high-integrity process can reset the sticky bit by running a special utility program provided by the protection system. The requirement of using a special utility program avoids the problem that other programs may accidentally reset the bit without the user intending to do it. This way, when a user clears the sticky bit, it is clear to the user that she is potentially raising the integrity of the file. The special utility program cannot be changed by low-integrity processes, so that its integrity level is always high.

Similar to the concept of RAP, we introduce file processing programs (FPP). A process running an FPP maintains its integrity level even after reading a low-integrity file. Programs that read a file's content and display the file on a terminal (e.g., vi, cat, etc.) need to be declared to be FPP.

We observe that our approach for handling file integrity is different from existing integrity models (such as Biba [21]), in which an object has one integrity level.

The integrity level of an object can be used to indicate two things: (1) the importance level of the object as a container (i.e., whether the object is used in some critical ways), and (2) the quality (i.e., trustworthiness, or, alternatively, contamination level) of information currently in the object. These two may not always be the same. When only one integrity level is used, one can keep track of only one of the two, which is problematic. Consider, for example, the system log files and the mail files. They are considered to be contaminated because they are written by processes who have communicated with the network. However, it is incorrect not to protect them, as an attacker who broke into the system through, say, httpd, would be able to change the log.

UMIP handles this by using a file's DAC permission to determine the importance level of the file, and using the sticky bit to track the contamination level. Even if a file has the sticky bit set (i.e., considered contaminated), as long as the file's DAC permission is not writable by the world, a low-integrity process still cannot write to the file (unless a policy exception exists). In other words, the set of write-protected files and the set of contaminated files intersect. This way, files such as system logs and mails are protected. This is different from other integrity models such as Biba, where once an object is contaminated, every subject can write to it. UMIP's design reduces the attack surface.

3.3.5 Files Owned by Normal User Accounts

Not all sensitive files are owned by a system account. For example, consider a user account that has been given privileges to sudo (superuser do) all programs. The startup script files for the account are sensitive. We follow the approach of using DAC info in MAC. If a file is not writable by the world, then it is write-protected. UMIP allows exceptions to be specified for specific users. Different users may have different exception policies. An account's exception policy may specify global exceptions that apply to all processes with that user's user-id. For example, a user may specify that a directory can be written by any low-integrity process and uses the directory to store all files from the network.

If the system administrator does not want to enable integrity protection for a user, so that the user can use the system transparently (i.e., without knowing the existence of UMIP), then the policy can specify a global exception for the home directory of the user with recursion so that all low-integrity processes with the user's user-id can access the user's files. We point out that even with such a global exception, UMIP still offers useful protection. First, the exception will be activated only if the process's effective user id is that user. Recall that we disallow a low-integrity process from using setuid to change its user id to another user account. This way, if an attacker breaks in through one daemon program owned by account A, the attacker cannot write to files owned by account B, even if a global exception for B is in place. Second, if the user is attacked while using a network client program, and the users' files are contaminated. These files will be marked by the sticky bit, and any process that later accesses them will drop its integrity level; the overall system integrity is still protected.

3.3.6 Other Integrity Models

The UMIP model borrows concepts from classical work on integrity models such as Biba [21] and LOMAC [22]. Here we discuss UMIP's novel features.

The Biba [21] model has five mandatory integrity policies: (1) the strict integrity policy, in which subject and object integrity labels never change; (2) the subject low-water mark policy, in which a subject's integrity level drops after reading a low-integrity object; (3) the object low-water mark policy, in which an object's integrity level drops after being written by a low-integrity subject; (4) the low-water mark integrity audit policy, which combines the previous two and allow the integrity levels of both subjects and objects to drop; (5) the ring policy, which allows subjects to read low-integrity objects while maintaining its integrity level. LOMAC [22] is an implementation of the subject low-water mark policy for operating systems. Each object is assigned an integrity level. Once assigned, an object's level never changes. It aims at protecting system integrity and places emphasis on usability. Compared with Biba and LOMAC, UMIP has the following novel features.

First, UMIP supports a number of ways to specify some programs as partially trusted to allow them to violate the default contamination rule or the default restrictions on low-integrity processes in some limited way. This enables one to use existing applications and administration practices, while limiting the attack surfaces exposed by such trust.

Second, in UMIP a file essentially has two integrity level values: whether it is protected and whether it is contaminated. The former is determined by the DAC permission, and does not change unless the file's permission changes. The latter is tracked using the sticky bit for protected files, and may change dynamically. The advantages of our approach is explained in Section 3.3.4.

Third, UMIP's integrity protection is compartmentalized by users. Even if one user has an exception policy that allows all low-integrity processes to access certain files owned by the user, another user's low-integrity process is forbidden from such access.

Fourth, UMIP allows low-integrity files to be upgraded to high-integrity. (This feature also exists in LOMAC.) This means that low-integrity information (such as files downloaded from the Internet) can flow into high-integrity objects (such as sys-

tem binaries); however, such upgrade must occur explicitly, i.e., by invoking a special program in a high-integrity channel to remove the sticky bit. Allowing such channels is necessary for patching and system ungrade.

Fifth, UMIP offers some confidentiality protection, in addition to integrity protection. For example, low-integrity processes are forbidden from reading files owned by a system account and not readable by the world.

Finally, UMIP uses DAC information to determine integrity and confidentiality labels for objects, whereas in LOMAC each installation requires manual specification of a mapping between existing files and integrity levels.

3.4 An Implementation under Linux

We have implemented the UMIP model in a prototype protection system for Linux, using the Linux Security Module (LSM) framework. We have been using evolving prototypes of the system within our group for a few months.

3.4.1 Implementation

The basic design of our protection system is as follows. Each process has a security label, which contains (among other fields) a field indicating whether the process's integrity level is high or low. When a process issues a request, it is authorized only when both the Linux DAC system and our protection system authorize it. A high-integrity process is not restricted by our protection system. A low-integrity process *by default* cannot perform any sensitive operation. Any exception to the above default policy must be specified in a policy file, which is loaded when the module starts.

The Policy Specification The policy file includes a list of entries. Each entry contains four fields: (1) a path that points to the program that the entry is associated with; (2) the type of a program, which includes three bits indicating whether the program is a remote administration point (RAP), a local service point (LSP), and

Table 3.1
The four forms of file exceptions in UMIP.

Syntax		Meaning
(f, read)	f is a regular file or a directory	Allowed to read f
(f, full)	f is a regular file or a directory	Allowed to do anything to f
(d, read, R)	d is a directory	Allowed to read any file in d recursively.
(d, full, R)	d is a directory.	Allowed to do anything to any file in d recursively.

a file processing point (FPP); (3) a list of exceptions; and (4) a list of executing relationships, which is a list of programs that can be executed by the current program with the exception policies enabled, even if the process is low integrity. If a program does not have a corresponding entry, the default policy is that the program is not an RAP, a LSP or an FPP, and the exception list and the executing relationship list are empty. An exception list consists of two parts, the capability exception list and the file exception list, corresponding to exceptions to the two categories of security critical operations. A file exception takes one of the four forms shown in the Table 3.1.

The authorization provided by file exceptions includes only two levels: read and full. We choose this design because of its simplicity. In this design, one cannot specify that a program can write a file, but not read. We believe that this is acceptable because system-related files that are read-sensitive are also write-sensitive. In other words, if the attacker can write to a file, then he can pose at least comparable damage to the system as he can also read the file. A policy of the form "(d, read, R)" is used in the situation that a daemon or a client program needs to read the configuration files in the directory d. A policy of the form "(d, full, R)" is used to define the working directories for programs.

3.4.2 Evaluation

We evaluate our design of the UMIP model and the implementation under Linux along the following dimensions: usability, security, and performance.

Usability One usability measure is transparency, which means not blocking legitimate accesses generated by normal system operations. Another measure is flexibility, which means that one can configure a system according to the security needs. A third usability measure is ease of configuration. Several features of UMIP contribute to a high level of usability: the use of existing DAC information, the existence of RAP, LAP, and FPP, and the use of familiar abstractions in the specification of policies. To experimentally evaluate the transparency and flexibility aspects, we established a server configured with Fedora Core 5 with kernel version 2.6.15, and enabled our protection system as a security module loaded during system boot. We installed some commonly used server applications (e.g., httpd, ftpd, samba, svn) and have been providing services to our research group over the last few months. The system works with a small and simple policy specification as given in Table 3.2. With this policy, we allow remote administration through the SSH daemon by declaring sshd as RAP. In this setting, one can also do remote administration through X over SSH tunneling and VNC over SSH tunneling. If one wants to allow remote administration through VNC without SSH tunneling, then he can declare the VNC Server as a RAP.

Security Most attack scenarios that exploit bugs in network-facing daemon programs or client programs can be readily prevented by our protection system. Successful exploitation of vulnerabilities in network-facing processes often results in a shell process spawned from the vulnerable process. After gaining shell access, the attacker typically tries downloading and installing attacking tools and rootkits. As these processes are low-integrity, the access to sensitive operations is limited to those allowed by the exception. Furthermore, if the attacker loads a shell or any other program, the new process has no exception privileges.

In our experiments, we use the NetCat tool to offer an interactive root shell to the attacker in the experiment. We execute NetCat in "listen" mode on the test machine as root. When the attacker connects to the listening port, NetCat spawns a shell process, which takes input from the attacker and also directs output to him. From

Table 3.2
A sample exception policy of UMIP

Services and Path of the Binary	Type	File Exceptions	Capability Exceptions	Executing Relationships
SSH Daemon /usr/sbin/sshd	RAP			
Automated Update: /usr/bin/yum	RAP			
/usr/bin/vim	FPP			
/usr/bin/cat	FPP			
FTP Server /usr/sbin/vsftpd	NONE	(/var/log/xferlog, full) (/etc/vsftpd, full, R) (/etc/shadow, read)	CAP_SYS_CHROOT CAP_SYS_SETUID CAP_SYS_SETGID CAP_NET_BIND_SERVICE	
Web Server /usr/sbin/httpd	NONE	(/var/log/httpd, full, R) (/etc/pki/tls, read, R) (/var/run/httpd.pid, full)		
Samba Server /usr/sbin/smbd	NONE	(/var/cache/samba, full, R) (/etc/samba, full, R) (/var/log/samba, full, R) (/var/run/smbd.pid, full)	CAP_SYS_RESOURCE CAP_SYS_SETUID CAP_SYS_SETGID CAP_NET_BIND_SERVICE CAP_DAC_OVERRIDE	
NetBIOS name server /usr/sbin/nmbd	NONE	(/var/log/samba, full, R) (/var/cache/samba, full, R)		
Version control server /usr/bin/svnserve	NONE	(/usr/local/svn, full, R)		
Name Server for NT /usr/sbin/winbindd	NONE	(/var/cache/samba, full, R) (/var/log/samba, full, R) (/etc/samba/secrets.tdb, full)		
SMTP Server /usr/sbin/sendmail	NONE	(/var/spool/mqueue, full, R) (/var/spool/clientmqueue, full, R) (/var/spool/mail, full, R) (/etc/mail, full, R) (/etc/aliases.db, read) (/var/log/mail, full, R) (/var/run/sendmail.pid, full)	CAP_NET_BIND_SERVICE	/usr/sbin/procmail
Mail Processor /usr/bin/procmail	NONE	(/var/spool/mail, full, R)		
NTP Daemon /usr/sbin/ntpd	NONE	(/var/lib/ntp, full, R) (/etc/ntp/keys, read)	CAP_SYS_TIME	
Printing Daemon /usr/sbin/cupsd	NONE	(/etc/cups/certs, full, R) (/var/log/cups, full, R) (/var/cache/cups, full, R) (/var/run/cups/certs, full R)	CAP_NET_BIND_SERVICE CAP_DAC_OVERRIDE	
System Log Daemon /usr/sbin/syslogd	NONE	(/var/log, full, R)		
NSF RPC Service /sbin/rpc.statd	NONE	(/var/lib/nfs/statd, full, R)		
IP Table /sbin/iptables	NONE		CAP_NET_ADMIN CAP_NET_RAW	

the root shell, we perform the following three attacks and compare what happens without our protection system with what happens when our protection system is enabled.

1. Installing a rootkit: rootkits can operate at two different levels. User-mode rootkits manipulate user-level operating system elements, altering existing binary executables or libraries. Kernel-mode rootkits manipulate the kernel of the operating system by loading a kernel module or manipulating the image of the running kernel's memory in the file system (/dev/kmem).

We use two methods to determine whether a system has been compromised after installing a rootkit. The first method is to try to use the rootkit and see whether it is successfully installed. The second method is to calculate the hash values for all the files (content, permission bits, last modified time) in the local file system before and after installing the rootkit. For the calculation we reboot the machine using an external operating system (e.g., from a CD) and mount the local file system. This ensures that the running kernel and the programs used in the calculation are clean. A comparison between the hash results can tell whether the system has been compromised.

We tried two well-known rootkits. The first one is Adore-ng, a kernel-mode rootkit that runs on Linux Kernel 2.2 / 2.4 / 2.6. It is installed by loading a malicious kernel module. The supported features include local root access, file hiding, process hiding, socket hiding, syslog filtering, and so on. Adore-ng also has a feature to replace an existing kernel module that is loaded during boot with the trojaned module, so that adore-ng is activated during boot. When our protection was not enabled, we were able to successfully install Adore-ng in the remote root shell and activat it. We were also able to replace any existing kernel module with the trojaned module so that the rootkit module would be automatically loaded during booting. When our protection system was enabled, the request to load the kernel module of Adore-ng from the remote root shell was denied, getting an "Operation not permitted" error. We got the same error when trying to replace the existing kernel module with the trojaned

module. When trying to use the rootkit, we received a response saying "Adore-ng not installed". We checked the system integrity using the methods described above. The result showed that the system remained clean.

The second is Linux Rootkit Family (LRK). It is a well-known user-mode rootkit and replaces a variety of existing system programs and introduce some new programs, to build a backdoor, to hide the attacker, and to provide other attacking tools. When our protection was not enabled, we were able to install the trojaned SSH daemon and replace the existing SSH daemon in the system. After that we successfully connected to the machine as root using a predefined password. When our protection was enabled, installation of the trojaned SSH daemon failed, getting the "Operation not permitted" error. The system remained clean.

2. Stealing the shadow File: Without our protection system, we were able to steal /etc/shadow by send an email with the file as an attachment, using the command "mutt -a /etc/shadow alice@haker.net < /dev/null". When our protection was enabled, the request to read the shadow file was denied, getting an error saying "/etc/shadow: unable to attach file" .

3. Altering user's web page files: Another common attack is to alter web files after getting into a web server. In our experiment, we put the user's web files in a sub directory of the user's home directory "/home/Alice/www/". That directory and all the files under the directory were set as not writable by the world. When our protection was enabled, from the remote root shell, we could not modify any web files in the directory "/home/Alice/www/". We could not create a new file in that directory. Our module successfully prevented user's protected files from being changed by low-integrity processes.

Performance We have conducted benchmarking tests to compare performance overhead incurred by our protection system. Our performance evaluation uses the Lmbench 3 benchmark and the Unixbench 4.1 benchmark suites. These microbench-

Table 3.3
The Unixbench 4.1 benchmark results of UMIP.

Benchmark	Base	Enforcing	Overhead (%)	SELinux(%)
Dhrystone	335.8	334.2	0.5	
Double-Precision	211.9	211.6	0.1	
Execl Throughput	616.6	608.3	1	5
File Copy 1K	474.0	454.2	4	5
File Copy 256B	364.0	344.1	5	10
File Copy 4K	507.5	490.4	3	2
Pipe Throughput	272.6	269.6	1	16
Process Creation	816.9	801.2	2	2
Shell Scripts	648.3	631.2	0.7	4
System Call	217.9	217.4	0.2	
Overall	446.6	435.0	3	

mark tests were used to determine the performance overhead incurred by the protection system for various process, file, and socket low-level operations.

We set up a PC configured with RedHat Linux Fedora Core 5, running on Intel Pentium M processor with 1400Hz, and having 120 GB hard drive and 1GB memory. Each test was performed with two different kernel configurations. The base kernel configuration corresponds to an unmodified Linux 2.6.11 kernel. The enforcing configuration corresponds to a Linux 2.6.11 kernel with our protection system loaded as a kernel module.

The test results are given in Table 3.3 and Table 3.4. We compare our performance result with SELinux. The performance data of SELinux is taken from [56]. For most benchmark results, the percentage overhead is small ($\leq 5\%$). The performance of our module is significantly better than the data for SELinux.

Table 3.4
The Lmbench 3 benchmark results of UMIP (in microseconds).

Microbenchmark	Base	Enforcing	Overhead (%)	SELinux(%)
syscall	0.6492	0.6492	0	
read	0.8483	1.0017	18	
write	0.7726	0.8981	16	
stat	2.8257	2.8682	1.5	28
fstat	1.0139	1.0182	0.4	
open/close	3.7906	4.0608	7	27
select on 500 fd's	21.7686	21.8458	0.3	
select on 500 tcp fd's	37.8027	37.9795	0.5	
signal handler installation	1.2346	1.2346	0	
signal handler overhead	2.3954	2.4079	0.5	
protection fault	0.3994	0.3872	-3	
pipe latency	6.4345	6.2065	-3	12
pipe bandwidth	1310.19 MB/sec	1292.54 MB/sec	7	
AF_UNIX sock stream latency	8.2	8.9418	9	19
AF_UNIX sock stream bandwidth	1472.10 MB/sec	1457.57 MB/sec	9	
fork+exit	116.5581	120.3478	3	1
fork+execve	484.3333	500.1818	3	3
for+/bin/sh-c	1413.25	1444.25	2	10
file write bandwidth	16997 KB/sec	16854 KB/sec	0.8	
pagefault	1.3288	1.3502	2	
UDP latency	14.4036	14.6798	2	15
TCP latency	17.1356	18.3555	7	9
RPC/udp latency	24.6433	24.8790	1	18
RPC/tcp latency	29.7117	32.4626	9	9
TCP/IP connection cost	64.5465	64.8352	1	9

4 TROJAN HORSE RESILIENT DISCRETIONARY ACCESS CONTROL

UMIP enhances DAC to protect the host integrity against the remote adversary. However, UMIP is still vulnerable to the Trojan horse attacks launched by the malicious local users because the local users are trusted in UMIP. In this chapter, we propose the IFEDAC model, which extends the UMIP model to defend against the Trojan horses and local exploits. We present the design and implementation details of the IFEDAC model.

4.1 An Overview of IFEDAC

One key concept in IFEDAC is the contamination source. Each contamination source represents a channel potentially controlled by a different entity who may compromise the system integrity. Each DAC user account that has a login shell and is not root is viewed as a separate contamination source. Remote network communication is another contamination source (denoted as net), which represents the remote attackers who do not have a local account. In the following description, we use subject and process interchangeably, and object and file interchangeably.

IFEDAC maintains an integrity level for each subject and object. The value of an integrity level is a set of contamination sources that indicate who may have gained control over the subject or who may have changed the content stored in the object. The integrity level is tracked using information flow technique, which is presented in Section 4.2.

In IFEDAC, the access control policy is specified by associating each object with a read protection class (*rpc*) and a write protection class (*wpc*); each is a set of contamination sources that indicates which entities are authorized to read from and write to the object. When a subject requests to read (write) an object, the access is

allowed if the subject's integrity level is a subset of the object's *rpc* (*wpc*), i.e., all of the contamination sources of the subject are allowed to access the object. In most cases, the *rpc* and *wpc* contain the entities who are authorized determined by the DAC policy.

Most real-world attacks are prevented by the default policy model of IFEDAC we have introduced so far. For example, if a remote attacker breaks in by exploiting a vulnerability in a network server, the server process controlled by the attacker will have net in the integrity level, and hence cannot access the system core files and the files that are authorized only to local users. Similarly, if a careless administrator executes a trojan horse downloaded from a malicious site or opens an email attachment that is a mal-formed file exploiting a vulnerable application, these files will have net in their integrity levels, as will the processes running and reading these files. Last, if a malicious local user u exploits a vulnerability in a setuid-root program to gain root privilege, the exploited process will have u in its integrity level, and hence it cannot access the system core files and other files that are not authorized to u.

While the default policy model of IFEDAC provides strong security guarantees to protect both core system files and user-owned files against trojan horses and vulnerability exploiting attacks, the model will disallow some legitimate operations, i.e., some processes need to access files that they are not authorized to access according to the policy. The same problem also exists in the DAC system and is handled by the setuid feature, which impose unlimited trust on these programs. IFEDAC handle this problem by specifying exceptions for programs. Exceptions imply trusts over programs and such trusts are strictly limited and can be clearly specified. We give a definition of exceptions in Section 4.2, analyze the underlying security assumptions for exceptions in Section 4.3, and discuss the exception policy configuration in practice in Section 4.4.

DAC offers adequate protection when all programs are benign and correct. IFEDAC can enforce the same policy without relying on this unrealistic assumption. We analyze the security properties IFEDAC can provide in Section 4.3. In short, IFEDAC

weakens the assumption to be (1) the programs that are explicitly identified as benign are benign (2) the programs that have exceptions are correct in processing input data.

4.2 The IFEDAC Model

We now give a formal definition of the IFEDAC model for Linux.

4.2.1 Elements in the IFEDAC Model

The IFEDAC model has the following elements:

- S denotes the set of all subjects (i.e., processes).
- O denotes the set of all objects (i.e., files).
- U denotes the set of users. The set $U \cup \{\mathsf{net}\}$ is the set of all contamination sources.

  The users are partitioned into two subsets: $U = A \cup N$, where A denotes system administrators and N denotes non-administrators. The users in A are trusted to perform system administration through certain limited channels, whereas the users in N are not. The administrators in A are similar to the notion of "sudoer" in the traditional DAC system in Linux.
- $\mathcal{L} = 2^{U \cup \{\mathsf{net}\}}$ is the set of all security labels that are used for integrity levels and protection classes. These labels form a lattice under a partial order $\geq$ such that $\ell_1 \geq \ell_2$ if and only if $\ell_1 \subseteq \ell_2$. The meet (i.e., greatest lower bound) of ℓ_1, ℓ_2 in the poset $\langle \mathcal{L}, \geq \rangle$ is therefore $\ell_1 \cup \ell_2$. The greatest element in $\langle \mathcal{L}, \geq \rangle$ is $\emptyset$, which we use $\top$ to denote; it means the subject or object has not been contaminated by any source. The least element is $U \cup \{\mathsf{net}\}$, which we use $\bot$ to denote.
- A function $int : S \cup O \to \mathcal{L}$ assigns an integrity level to each subject and each object.

- Each object o has three protection classes.
  - A function $rpc : O \rightarrow \mathcal{L}$ assigns a *read protection class* to each object.

    The value $rpc(o)$ can be explicitly set. If it is not explicitly set, then $rpc(o)$ is inferred from DAC as follows: If o is world-readable, then $rpc(o) = \bot$. Otherwise, $rpc(o) = U_r(o)$, where $U_r(o)$ is the set of users in U who are authorized to read o. If o is group-readable, then $U_r(o)$ may change when group membership changes. IFEDAC uses the group membership information at the time of access to determine $rpc(o)$.
  - A function $wpc : O \rightarrow \mathcal{L}$ assigns a *write protection class* to each object.

    Unless explicitly set, the value $wpc(o)$ is inferred from DAC similarly to $rpc(o)$. We expect that for the vast majority of objects, $rpc(o)$ and $wpc(o)$ are inferred from DAC.
  - A function $apc : O \rightarrow \mathcal{L}$ assigns an *admin protection class* to each object.

    This function determines which subject can change the $rpc(o)$ and $wpc(o)$ either directly or indirectly, through changing its DAC permission bits. As the Linux DAC mechanism allows only root or the owner of an object to change the permission bits, in IFEDAC we choose $apc(o) = \{\mathsf{owner}(o)\}$.
- A function $spc : S \rightarrow \mathcal{L}$ assigns a *subject protection class* to each subject. The value $spc(s)$ determines which subjects can send signals or use ptrace to interrupt or control the subject s. The rules for determining spc is described in next subsection.

4.2.2 Access Control Rules in IFEDAC

IFEDAC has 17 rules for access control and label maintenance(See Table 4.1). They are separated into four parts: subject integrity tracking, object integrity tracking, file system protection, and inter-process communications (IPC) protection. These

Table 4.1
The 17 access control and label maintenance rules of IFEDAC. The last column indicate whether the rule can have an exception.

	condition or effect		exceptions
Subject Integrity Tracking			
After creating the first subject s_0	$int(s_0) \leftarrow \top$	(m1)	no
After s creates s'	$int(s') \leftarrow int(s)$	(m2)	no
After s executes o	$int(s) \leftarrow int(s) \cup int(o)$	(m3)	no
After s reads from the network	$int(s) \leftarrow int(s) \cup \{\mathsf{net}\}$	(m4)	yes
After s reads from o	$int(s) \leftarrow int(s) \cup int(o)$	(m5)	yes
After s logs in a non-administrator u	$int(s) \leftarrow int(s) \cup \{u\}$	(m6)	no
After s_1 receives IPC data from s_2	$int(s_1) \leftarrow int(s_1) \cup int(s_2)$	(m7)	yes
Object Integrity Tracking			
When o is created by s	$int(o) \leftarrow int(s)$	(o1)	no
When $int(o)$ is not previously assigned	$int(o) \leftarrow wpc(o)$	(o2)	no
After o is written by s	$int(o) \leftarrow int(s) \cup int(o)$	(o3)	yes
Object Protection			
For s to read o	require $int(s) \geq rpc(o)$	(a1)	yes
For s to write o	require $int(s) \geq wpc(o)$	(a2)	yes
For s to change $rpc(o)$ or $wpc(o)$	require $int(s) \geq apc(o)$	(a3)	yes
For s to change $apc(o)$	require $int(s) \geq \top$	(a4)	no
For s to change $int(o)$ to ℓ'	require $int(s) \geq apc(o) \wedge int(s) \geq \ell'$	(a5)	no
IPC Protection			
For s_1 to interrupt s_2	require $int(s_1) \geq spc(s_2)$	(i1)	no
For s_1 to ptrace s_2	require $int(s_1) \geq spc(s_2) \wedge int(s_1) \geq int(s_2)$[a]	(i2)	no

[a] If $int(s_1) \neq \top$, s_2 cannot have any exception privileges. All these conditions should be satisfied during the whole tracing period. A violation will stop the tracing.

rules are summarized in Table 4.1. Some of these rules can have exceptions. We describe them in Section 4.2.3. We point out that end users do not need to know these rules to use a Linux system with IFEDAC, just as they do not need to know the intricacies of setuid-related system calls to use current Linux.

Subject Integrity Tracking The subject's integrity level is determined as follows.

- (m1). The first process, init, has integrity level $\top$.

- (m2). When a new process is created, it inherits the parent process's integrit level.

- (m4). When a process receives network traffic, its integrity level is updated to include net as an additional contamination source. This represents that the attacker who controls the network may have gained control over the subject by exploiting vulnerabilities in the subject.

- (m3), (m5). When a process s executes or reads an object o, its integrity level is contaminated by o, that is, $int(s) \leftarrow int(s) \cup int(o)$. This represents whichever source that may have contaminated the object o now may have gained control over s[1].

- (m6). When a process logs in a user u, the process is contaminated according to the type of the user. If u is an administrator, then the process's integrity level remains unchanged. Otherwise, if u is a non-administrator, the process's integrity level is updated to include u as an additional contamination source, which represents that the user u has gained control over the process. We use the fact that a login in Linux triggers an event wherein all three uids (real uid, effective uid, saved uid) of a process are changed to a new user. This event occurs whether the login is through a terminal and the X desktop (the "login" process), via an ssh or ftp server, or by the execution of the "su" command.

Object Integrity Tracking For subject integrity tracking to be effective, we also need object integrity tracking.

- (o1). When a new object is created by a process, the object's integrity level is initialized to be the process's integrity level.

[1]In our subject integrity racking rules, the integrity levels of processes can only go down and can never go up. One may worry that the whole system converges to $\perp$ after a time. This is not the case. It is true that the integrity level of any individual process can never go up after it goes down. However, as some processes, e.g., the root process of the system (i.e., init), are high, there are always new processes coming up that are also high. One can use a tree as an analogy. Once a leaf is dead, it does not become alive again. However, because the root is alive, new leaves keep coming up.

- (o2). For an object that is created before IFEDAC is deployed, its integrity level is initialized as its write protection class, because the write protection class is derived from the DAC permissions which can indicate how the object is protected before deploying IFEDAC.

- (o3). When an object o is modified by a process s, the object's integrity level is contaminated by s, that is $int(o) \leftarrow int(o) \cup int(s)$.

File System Protection The access control rules for file system protection are as follows.

- (a1), (a2). For a subject s to read o, we require that $int(s) \geq rpc(o)$. Similarly, we require that $int(s) \geq wpc(o)$ for s to write to o.

- (a3). For s to change the *rpc* (o) and *wpc* (o), we require $int(s) \geq apc(o)$.

- (a4). Linux DAC allows only root to change the owner of a file; thus IFEDAC adopts the policy that for s to change $apc(o)$, we require $int(s) = \top$.

- (a5). An object's integrity level can be updated explicitly. This is necessary, for example, to allow system updates. The integrity level of downloaded updates will include net, and needs to be upgraded to $\top$ before the updates can be installed. However, for s to update o's integrity level to ℓ, we require both $int(s) \geq apc(o)$ and $int(s) \geq \ell$. The former requires that s represents the owner of o, and the latter prevents malicious upgrading beyond one's own integrity level.

Inter-process Communications Modern Linux supports various mechanisms for inter-process communication (IPC). IFEDAC handles IPC by categorizing the IPC mechanisms into three types.

- (m7). We call the first type *Data Sending*. The IPC mechanisms that belong to this type include pipes, FIFO, message queues, shared memory, local sockets

and loopback network communication. They can be used to send free-formed data, and such data can be crafted to exploit bugs in the receiving process. Therefore after $s1$ receives IPC traffic from $s2$, $int(s1) \leftarrow int(s1) \cup int(s2)$. IFEDAC does not apply additional control to these IPCs, because they require active participation of both the sender and the receiver. Without the receiver's active participation, the sender cannot force the receiver to receive data.

- (i1). We call the second type *Interrupting*. The IPC mechanisms that belong to this type include sending signals and changing scheduling parameters of another process (through the sys_set_priority() and sys_set_scheduler() system calls). For most signals, the default behavior of the receiving process is to terminate, core-dump, or stop, unless the process registers its own signal handlers to overwrite the default actions. We do not want the attacker to be able to terminate a critical system service or change the execution state of a process that belongs to another user. In other words, a user can only interrupt his own processes and only system administrators can do so to the system processes. IFEDAC achieves that by defining the subject protection class to indicate the "owner" of a process. The *spc* value is determined as follows. Initially when a new process is created, it inherits the parent process's protection class. When a process logs in a user u, its protection class is updated to $\{u\}$. Then, for s_1 to deliver an interrupting IPC to s_2, we require $int(s_1) \geq spc(s_2)$. If succeed, unlike the data sending IPCs, the receiver's integrity level does not change because it is difficult to use signaling to exploit a vulnerable process.

- (i2). We call the third type *Controlling*. The only IPC mechanism that belongs to this type is ptrace. It enables the tracing process to observe and control the traced process and is used primarily for debugging. The tracing process can arbitrarily manipulate the memory and registers of the traced process, and even inject code into the traced process. As with interrupting IPCs, IFEDAC requires $int(s_1) \geq spc(s_2)$ for s_1 to ptrace s_2. In addition, because the tracing process

can easily abuse the privileges of the traced process, IFEDAC requires s_2 does not have any privileges that are not available to s_1. That is, $int(s_1) \geq int(s_2)$, and s_2 does not have any exceptions if $int(s_1) \neq \top$. These conditions should be satisfied during the whole tracing period. Any violation will stop the tracing immediately.

4.2.3 Exceptions to the Rules

The information flow tracking is a restricted enforcing mechanism and the default policy described above would break some applications that need to access the files that they are not authorized to access. The same problem also exists in the DAC system and is handled by the setuid feature, which impose unlimited trust on the setuid programs. IFEDAC handle this problem by introducing exceptions. The exceptions are associated with program binaries, and imply that these programs are trusted in certain ways. When a program binary that has exceptions is loaded (through the execve system call), if the current process's integrity level satisfies the minimal integrity restriction and the program binary has the integrity level $\top$, the exceptions are enabled. Once a new binary is loaded, the old exceptions are gone.

Exceptions to the subject integrity tracking Exception to the network contamination rule ((m4) in Table 4.1) is by the notion of a *remote administration point (RAP)*. A process running a RAP program maintains its integrity level when receiving network traffic. If one wants to allow remote system administration through, for example, the secure shell daemon, then one can identify the SSH daemon as a RAP. The trust assumption underlying a RAP declaration is that when the program is started in a benign environment it will process the network input correctly and the attacker cannot gain control of it by sending malformed network packets. We stress that whether to declare a program as RAP is a decision made by the local system administrator.

Similarly, exceptions to the file reading and IPC contamination rules ((m5) and (m7) in Table 4.1) are done under the notion of a *local service point (LSP)*. The process running a LSP program maintains its integrity level when reading from files or receiving IPC data from other processes. The trust assumption underlying the LSP declaration is that the program will process file and IPC input correctly.

The concepts of RAP and LSP are similar to the ring policy in the Biba model, in which a subject can read objects of an arbitrary integrity level without dropping its own integrity level. This is also similar to the notion of well-formed transactions in the Clark-Wilson model, which can read low integrity unconstrained data items and write to high integrity constrained data items.

Exceptions to object protection and integrity tracking For some programs, the integrity level at which it is normally running does not dominate the protection class of some objects it needs to access. For example, the ftp daemon will be running at the integrity level $\{\mathsf{net}\}$, but it needs to read from the /etc/shadow file to authenticate users. However, the shadow file has the read protection class $\top$, and thus the default policy will stop the access. We deal with this by allowing exceptions to object protection rules ((a1) and (a2) in Table 4.1). One can specify a set of file access exceptions for a program. Each exception enables a process running the program to read from or write to a file while violating the object protection rules. For the example of ftp server, one can specify the ftp daemon program to have a file access exception to read from the file /etc/shadow.

A file write exception contains an additional field to enable an exception to the object integrity tracking rule ((o3) in Table 4.1). When that field is set, after the program writes to the file, the file's integrity level remains unchanged. For example, the program /etc/passwd needs an exception to write to the file /etc/shadow when it is executed by a non-administrator u at the integrity level $\{u\}$. Moreover, the shadow file's integrity level should remain as $\top$ after being modified by passwd.

Note that since the old exception privileges are gone after a new binary is loaded, even if a vulnerable program is granted some exception privileges and the process running that program is exploited by the attacker, a shell (or other programs) spawned from the exploited process won't have any exception privileges.

4.3 Security Properties of IFEDAC

Recall that for DAC to be effective, all programs need to be assumed to be benign and correct. By introducing information flow techniques, IFEDAC aims at weakening this unrealistic assumption. We now analyze what the security properties IFEDAC can provide and what are the necessary assumptions to achieve them. The high-level security goal of IFEDAC is that confidentiality and integrity properties of a system are preserved under attacks.

4.3.1 Defining Integrity

Defining integrity in the context of operating systems is a difficult task. One can start by defining integrity as the property that key components do not change. This definition is too strong, as key files (e.g., /etc/shadow) and the kernel data structures need to change. As key components must change, one may modify the property to state that the resulting state after a change must satisfy certain constraints that can be precisely specified and checked. However, it is infeasible (and often impossible) to characterize these constraints. Next, one could refine the definition of integrity as the property that key components are changed only through certain programs. This property, though, is insufficient. Text editors must be allowed to modify key system script files. Yet, one cannot say these files have integrity solely because all updates are performed only through these editors. Finally, one can define integrity by declaring that key components are changed only by certain users. We believe this last choice most accurately reflects the intuition. If the change is intended by authorized users, then integrity is preserved; otherwise, it is violated.

We thus define integrity informally as

Integrity means all updates reflect authorized users' intentions.

To formalize this, we must identify two things: (1) who is authorized to perform an update, and (2) whose intention a subject (process) reflects. In IFEDAC, the former is specified by the write and administration protection classes. Any user in $wpc(o)$ (as well as root) is authorized to update o. For the latter, we observe that an integrity label has a natural interpretation as a representation of intentions. A label of $\top$ means the intention of the root user. A label of $\{u_1, u_2\}$ means the intention of u_1, u_2, or root. If we have $int(s) = \{u_1, u_2\}$ and $wpc(o) = \{u_1, u_3\}$, then s cannot update o, because the update may reflect the intention of u_2, who is not authorized to do so.

Therefore, a key property we need to show is that IFEDAC maintains the integrity levels for subjects correctly. That is, if a subject has integrity level ℓ according to IFEDAC, then the subject is *benign for integrity level* ℓ in the sense that any operation performed by the subject reflects the intention of only those users in ℓ.

To achieve this goal, we start by noting that integrity protection requires some degree of trust that programs do not introduce bad data. We can contrast this with confidentiality protection, for which if an untrusted subject never reads any secret information, it can not later write or leak secret information. For integrity, it is not enough to control what the subject reads, as it can create bad data without reading bad data. This observation suggests that integrity is not simply an information flow property. The strict integrity policy in the Biba model allows a subject at integrity level ℓ to read objects at ℓ or higher and write objects at level ℓ or lower. This implicitly requires that one trusts a subject at integrity level ℓ to be able to generate data at integrity level ℓ when reading data only at level ℓ or higher. Therefore, the code executed in the subject must be both functional and not malicious for integrity level ℓ. We say such a program is assumed to be *benign* for integrity level ℓ. Intuitively, the behavior of a benign program reflects the users' intention. For example, the basic utilities on a system such as editors and file manipulation tools are considered benign,

not because they cannot be used to do bad things, but because they reflect the users' intentions.

We still need to translate the benign property of a static program file to the benign property of a running process. To do this, we assume the following axiom.

Axiom 1 *If a program is benign for an integrity level ℓ, then when it is executed by a subject that is benign at integrity level ℓ or higher, and the subject reads only input at integrity level ℓ or higher, the subject is benign for integrity level ℓ.*

We note that assuming that a program is benign is a weaker assumption than that the program is both benign and correct. A benign program is not trusted to handle malicious input. In short, a benign program mostly works as expected. But when it is exposed to malicious input, it may not do so anymore.

4.3.2 Integrity Protection Properties

We now show that IFEDAC achieves the integrity goal that all updates reflect authorized users' intentions, under a number of assumptions. It suffices to show that IFEDAC maintains the following three invariants: (1) Every subject with integrity level ℓ is benign for that integrity level. (2) The content of every file with integrity level ℓ is only controlled by the users in ℓ. (3) For every file o, $wpc(o)$ correctly identifies the authorized users.

These are maintained by IFEDAC under the following assumptions. (1) When IFEDAC is enabled, the integrity levels of files are correct. For example, a program labeled with integrity level ℓ is benign for that level. (2) When IFEDAC is enabled, files are labeled with the correct write and administration protection classes. (3) The hardware has not been compromised. (4) The kernel and the programs that have exceptions are trusted either to process input correctly or not to fail in a way that the attacker can directly exploit the exceptions. (5) When a legitimate user *intends to* upgrade a file's integrity level, the decision is correct. When a legitimate user *intends to change* the write or admin protection class of an object, the decision is correct.

Assumptions (1) and (2) say that the initial labels are correct. IFEDAC cannot defend against physical attacks such as changing the BIOS settings to boot from the attacker's media; hence assumption (3). Assumption (5) means that the system must trust the legitimate user's intentions. Rather than assuming all programs are benign, assumptions (1) and (5) indicate that IFEDAC requires only the programs that are explicitly identified as benign (by setting the program's integrity level) to be benign.

Assumption (4) requires more examination. First, as IFEDAC works within the kernel, we must assume the kernel has no vulnerabilities the attacker can exploit. This assumption is also needed for similar protection systems, such as Security Enhanced Linux (SELinux) or AppArmor. IFEDAC extends this assumption so that a process running a program specified as RAP cannot be compromised by receiving network traffic, as the program is assumed to process network data correctly. Similarly, any program specified as LSP is assumed to process IPC inputs correctly. Read exceptions do not affect integrity, as it does not involve an update. If a program has a write exception, it is assumed that (1) the program correctly handles bad input (similar to the previous discussion of RAP and LSP), or (2) if the program is exploited, the attacker is unable to inject malicious code directly into the address space to take advantage of the exception. In a typical exploit, the attacker injects the shell code into the vulnerable process, then runs malicious code in the spawned shell. Under IFEDAC, the spawned shell loses the write exceptions. The other possibility is for the attacker to inject all of the malicious code directly into the address space, but this task is more difficult than getting a shell, and is more easily defended against (e.g., with a non-executable stack).

Almost all exceptions we have are also allowed in the SELinux Targeted policy. Each of our exception specifications makes the underlying security assumption explicit, which is not the case in, for example, SELinux.

4.3.3 Confidentiality Protection in IFEDAC

As in integrity protection, DAC assumes all programs to be benign for confidentiality. For example, when one uses /bin/cat to view a file's contents, one implicitly trusts that it will not secretly send the file through an email, or create a world-readable copy of the file. Some programs will also write to, for example, files readable by others while reading files readable only by the user. Those programs are trusted to correctly declassify information. In IFEDAC, if we assume that a subject that is benign at integrity level ℓ can correctly declassify information at level ℓ, then confidentiality is also preserved by IFEDAC, under similar assumptions for integrity protection. Of course, we need to assume that the initial read protection classes of objects are set correctly. Also, when a program has a read exception, we assume that the program either (1) can handle malicious input, or (2) cannot be exploited in a way that the attacker injects malicious code into the address space and takes advantage of the exceptions.

4.4 Deployment and Usability

We established a server and a personal workstation with the IFEDAC module loaded during system boot. On the server machine, we installed some commonly used server applications (e.g., httpd, ftpd, samba, svn) and provided services to our research group. Multiple user accounts exist on the server, some of which are allowed to perform system administration (specified as a sudoer and a member of A, the set of administrators). On the personal workstation, we perform everyday jobs on the Gnome desktop. The jobs we tested include web browsing, emailing, file downloading, instant-messaging and normal system administration. We report some interesting experiences of deploying, configuring and using the IFEDAC module.

A Usage Case We use the email client ThunderBird as a usage case to describe how to configure and use IFEDAC in practise. When a local user u launches the application

of ThunderBird, the process inherits the parent's integrity level and runs at $\{u\}$. After the process receives network traffic from remote servers, its integrity level is updated to $\{u, \mathsf{net}\}$. The process needs to read from and write to the configuration files and the files storing the downloaded messages, which are located in the directory \$HOME/.thunderbird by default (\$HOME refers to u's home directory). In DAC, those files are writable only by u; hence in IFEDAC they have the write protection class $\{u\}$, which is higher than the process's integrity level. To enable the access, we grant the binary executable of ThunderBird an exception privilege to read from and write to the directory \$HOME/.thunderbird/ recursively. In this way, the email client can function normally.

If the user wants to save a file from an email attachment to the file system, this is achieved by the *Internet Directory*. The user u can create an Internet directory and set its write protection class to be $\{u, \mathsf{net}\}$. When he wants to save an email attachment, he first saves the file to the Internet directory. The saved file's integrity is initialized as the process's integrity level, $\{u, \mathsf{net}\}$, which can be manually upgraded later if the user has confidence in the file and wants to use it with a higher integrity level. The Internet Directory is not only used by the email client; in fact the user may create multiple Internet directories and can store all downloaded files (e.g., through a web browser, ftp client, instant messenger) to those directories and later upgrade their integrity levels if he wants to. The Internet directory is an example where the write protection class is lower than that inferred from DAC permission. In DAC, that directory is writable only by the owner.

Possible attack channels exposed by the email client include executing a mal-ware in an email attachment, opening an attachment that is a mal-formed file exploiting a vulnerable application and a vulnerability in the email client being exploited by a remote attacker. In all these attacks, the process controlled by the attacker will have the integrity level $\{u, \mathsf{net}\}$ and can only access the files writable by the world and the user's Internet directories. See the security evaluation for details about testing against attacks.

System Administration and Automatic Update Many modern Linux systems allow normal user accounts to perform system administration through the sudo tool. One benefit is better accountability. With IFEDAC we can still use this common usage practise with better security property. These accounts should be in A, the set of administrators. Even though users in A are trusted, each of them still corresponds to a contamination source. This separation helps to enforce the DAC policy. Additionally, most tasks these users perform are user-level jobs that do not need full privileges. Viewing these users as separate contamination sources limits any errors made for a user-level job to that particular user.

For a user $u \in A$, most of his files have the write protection class and integrity level at $\{u\}$ or lower, except for some startup files (e.g., the startup script of the shell) that are used during login. When making u an administrative user, one upgrades the write protection class and integrity level of the user's startup files to $\top$. For example, the startup scripts for Bash Shell include: /.bash_rc, /.bash_profile and /.bash_logout. When he logs in, he gets a shell at $\top$, where he can perform system administration tasks. However, any descendant process that reads his normal files will drop to the integrity level $\{u\}$. He can also downgrade the shell's integrity level to $\{u\}$ by executing a utility program provided by IFEDAC, when he starts performing user-level jobs. To perform system administration later, he needs to obtain a fresh channel with at $\top$ by logging in again.

The startup files owned by users in A provide an example where the write protection class is higher than that inferred from DAC permissions. In DAC, those files are owned by normal users, rather than root. Assigning those files with the write protection class $\top$ helps protecting system integrity, because those files are critical and should only be modified at level $\top$.

Remote administration through a secure shell daemon is expected in some situations. As mentioned in Section 4.2, one can allow that by specifying the program /usr/sbin/sshd to be a remote administration point (RAP). Also, automatic updates are commonly used in today's commercial operating systems. These programs down-

load updated packages and automatically install them. To enable automatic updates in IFEDAC, the administrator can specify the update program as a RAP, trusting that it is not vulnerable. For example, the automatic update programs in Fedora Core include /usr/bin/yum and /usr/share/rhn/rhn_applet/applet.py.

Exception Policy Configuration Most programs can work with IFEDAC without any modification and policy configuration. Two kinds of programs need exceptions in IFEDAC: network programs and setuid root programs.

Network Programs Like the email client described before, network programs run at the integrity level $\{\mathsf{net}\}$ or $\{u, \mathsf{net}\}$, but need to access configuration and log files that have higher protection class. See Table 4.2 for a sample policy for some commonly used server and client programs. For each program, only a small number of exception privileges are needed. The policy can be easily understood.

Setuid Root Programs The setuid-root programs run at integrity levels $\{u\}$ when they are executed by a non-administrator u. The default policy will forbid them from performing system critical operations that require the integrity level $\top$. However, most of these programs need to perform such high-integrity tasks. A sample exception policy for setuid-root programs and network programs in Fedora Core 5 is shown in Table 4.3. Those exceptions will be activated only from an integrity level $\{u\}$. That is, if a process has integrity level $\{u_1, u_2\}$ or $\{u_1, \mathsf{net}\}$, it does not get any exceptions when loading the setuid root programs.

IFEDAC provides better protection for setuid-root programs than DAC in three aspects. First, in IFEDAC the privileges gained by those programs are restricted based on the least privilege principle. For example, the program "ping" needs to be setuid-root only because it performs raw socket operations (controlled by the capability CAP_NET_RAW). IFEDAC grants only that exception privilege to "ping", whereas DAC allows "ping" to perform any critical operation. IFEDAC significantly reduces the damage caused by an exploit in "ping". Second, the shell spawned from an exploit loses the exception privileges. In order to abuse the exception privileges,

the attacker must inject all malicious code into the address space of the vulnerable process, which is more difficult. Third, with IFEDAC, only malicious local users are able to take advantage of buggy setuid-root programs. Remote attackers breaking in through network programs cannot use setuid-root program to elevate their privileges, because they cannot use the exception privileges if the integrity level contains net.

What end-users need to know about IFEDAC? In practice, the exception policies should be specified and distributed by the software and OS vendor (e.g., included in the installation packages). System administrators only need to make high-level decisions such as whether to allow remote administration or not. Similarly, administrator only need to specify which users are allowed to perform system administration; the configurations are done automatically by the system. Normal users should understand the basic meaning of read protection class and write protection class for objects (which are similar to ACL). In most situations, the protection classes are derived from the DAC policy and configuring them are achieved by changing the permission bits. In our experiments, the only case that a normal user need to explicitly manage the protection class is to setup the Internet directory, which can be done automatically by the system when a new user is created. Normal users should also understand the integrity level for objects and, in a few situations, users need to manually upgrade an object's integrity level. For example, when a user wants to use a downloaded program to manage his own files, he need to upgrade the program's integrity level from $\{u, \mathsf{net}\}$ to $\{u\}$.

4.5 Discussion

Compare IFEDAC with Traditional DAC Compared with traditional DAC, IFEDAC provides much stronger security guarantee. The traditional DAC has two goals: (1) protect system resources from local users. Both malicious behavior and careless mistakes performed by local users won't compromise the system. (2) provide user separation. User-owned resources are protected against other malicious users.

Table 4.2
Exception privileges for network programs in IFEDAC

Programs	File Exceptions	Capability Exceptions
Servers		
FTP Server /usr/sbin/vsftpd	read /etc/shadow; write to /etc/vsftpd, /var/log/xferlog;	CAP_SYS_CHROOT CAP_NET_BIND_SERVICE
Web Server, /usr/sbin/httpd	read /etc/pki/tls, /var/www; write to /var/log/httpd, /var/run/httpd.pid	
Samba Server /usr/sbin/smbd	write to /var/cache/samba, /etc/samba, /var/log/samba, /var/run/smbd.pid	CAP_SYS_RESOURCE CAP_NET_BIND_SERVICE
NetBIOS Server, /usr/sbin/nmbd	write to /var/cache/samba, /var/log/samba	
Version Control Server /usr/bin/svnserve	write to /usr/local/svn	
SMTP Server /usr/sbin/sendmail	read /etc/aliases.db; write to /var/spool/mqueue, /var/spool/mail, /var/spool/clientmqueue, /etc/mail, /var/log/mail	CAP_NET_BIND_SERVICE
Clients		
Browser, /usr/lib/.../firefox-bin	write to /tmp, $HOME/.mozilla/firefox	
Email, /usr/lib/.../thunderbird	write to /tmp, $HOME/.thunderbird	

In achieving the two security goals, traditional DAC makes a strong assumption: all programs are benign and correct. This assumption is far from the reality today due to the large amount of malware and software vulnerabilities. IFEDAC achieves the same security goals as the traditional DAC by enforcing the discretional policy specified in the existing DAC system. However, IFEDAC eliminates the unrealistic trust over the programs. By using the information flow techniques to track the principal of a running process, IFEDAC is able to defend against Trojan horse and vulnerability exploitation.

In terms of usability, certainly the policy enforcement mechanism in IFEDAC is more complicated than that in traditional DAC. However, the end users generally do not need to understand or even know about those rules. Most of IFDEAC policy is derived from the existing DAC policy, the user can specify IFEDAC policy in the

Table 4.3
Exception privileges for setuid-root program in IFEDAC

Usage Types	Setuid Root Programs	Exception Privileges [a] [b],
User information updates	passwd, chage: change user password and expiry information	create files in /etc; write to /etc/passwd, /etc/shadow;
	chsh, chfn: change user login shell and finger information	create files in /etc; write to /etc/passwd;
PAM (Pluggable Authentication Module) utilities	unix_chkpwd: check user password	read /etc/shadow
	userhelper: update user information	create files in /etc; write to /etc/passwd, /etc/shadow; read from /var/run/sudo;
Group configuration	gpasswd	create files in /etc; write to /etc/gshadow, /etc/group, /var/run/utmp;
User identity switches	newgrp: login to a new group	read /etc/group, /etc/passwd, /etc/shadow, /etc/gshadow; write to /var/run/utmp
	su, sudo, sudoedit: run a shell or other commands as another user	read /etc/shadow, /etc/sudoer
Network utilities	ping, ping6: ping network hosts	use CAP_NET_RAW
Mounting utilities	mount, umount	create files in /etc; read /etc/fstab; write to /etc/mtab, /etc/filesystems;
r-commands	rlogin, rcp, rsh: remote login, copy and shell	write to /etc/krb5.conf, /etc/krb.conf; use CAP_NET_BIND_SERVICE
Job scheduling	at: schedule a command	write to /var/spool/at, /var/run/utmp; read /etc/at.allow, /etc/at.deny
	crontab: edit the regular job schedule	write to /var/spool/cron; read /etc/cron.allow, /etc/cron.deny;

[a] The write privilege over a file infers the read privilege over the same file.
[b] All write exceptions keep the integrity level of the written files.

same ways as they administer traditional DAC systems, which is believed to be easy and intuitive. Information flow tracking is a more restricted enforcing mechanism and would break some applications that can run in traditional DAC. IFEDAC address this issue by introducing exceptions to the default policy. The fundamental concept is similar to the setuid feature in traditional DAC. However, as discussed in Section 4.3,

the assumptions implied by exceptions are strictly limited and the attack surface exposed by the trusted programs are significantly reduced.

Compare IFEDAC with MAC The traditional approach to address the weakness of DAC is to build a MAC system on top of the existing DAC system. Several MAC systems have been deployed in real-world commercial operating systems, such as SELinux [11] and AppArmor [9]. Such MAC systems are flexible and powerful. Through proper configuration, they could result in highly-secure systems. However, they are also complex and intimidating to configure. For example, SELinux has 29 different classes of objects, hundreds of possible operations, and thousands of policy rules for a typical system. The SELinux policy interface is daunting even for security experts. Besides the complexity in policy configuration, the MAC systems are also difficult to use when configured with a policy providing comprehensive security. For example, the strict policy shipped with SELinux in Fedora Core 2 used a disallow-by-default approach. The policy has to be kept updated with every change to the operating system. In particular, any newly installed application won't operate without specifying a policy.

To overcome the usability issues, the policy actually enforced in the real-world MAC systems only confine tens of sever daemons and about a dozen of setuid-root programs. All other programs remain unconfined and are trusted to be benign and correct. Exploiting any of those programs would lead to system compromise. In addition, the real-world policy make the MAC systems vulnerable to Trojan horse, because all new installed applications that do not have corresponding policies are treated as unconfined.

5 PRETTY-BAD-PROXY: AN OVERLOOKED ADVERSARY IN BROWSERS' HTTPS DEPLOYMENT

The DAC mechanism used in operating systems is vulnerable to vulnerability exploits and Trojan horses because DAC fails to identify the true origins of the processes. Similar design mistakes occur in the same-origin policy (SoP) model implemented in major browsers. In this chapter, we discuss the pretty-bad-proxy adversary against HTTPS and present the vulnerabilities we discovered in the HTTPS deployments in major browsers. In the next chapter, we discuss the cross-site request forgery attack and propose a browser-side defense mechanism.

5.1 Motivation and Overview

HTTPS is an end-to-end cryptographic protocol for securing web traffic over insecure networks. This work is motivated by our curiosity about whether the same adversary that is carefully considered in the design of HTTPS is also rigorously examined when HTTPS is integrated into the browser. In particular, we focus on an adversary called "Pretty-Bad-Proxy" (PBP), which is a man-in-the-middle attacker that specifically targets the browser's rendering modules above the HTTP/HTTPS layer in order to break the end-to-end security of HTTPS. Figure 5.1 illustrates this adversary: PBP can access the raw traffic of the browser (encrypted and unencrypted), but it is unable to decrypt the encrypted data on the network. Instead, the PBP's strategy is to send malicious contents through the unencrypted channel into the rendering modules, attempting to access/forge sensitive data (which flow in the encrypted channel on the network) above the cryptography of HTTPS.

With a focused examination of the PBP adversary against various browser behaviors, we realize that PBP is indeed a threat to the effectiveness of HTTPS deploy-

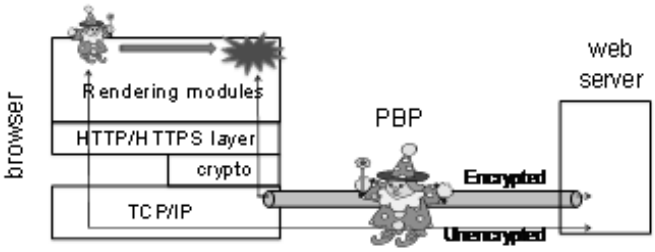

Figure 5.1. PBP attacks the encrypted data after they are decrypted above the HTTPS layer

ments. We have discovered a set of PBP-exploitable vulnerabilities in IE, Firefox, Opera, Chrome browsers and many websites. They are due to a number of subtle behaviors of the HTML engine, the scripting engine, the HTTP proxying, and the cookie management. By exploiting the vulnerabilities, a PBP can obtain the sensitive data from the HTTPS server. It can also certify malicious web pages and impersonate authenticated users to access the HTTPS server. Although all attacks fool the HTTP/HTTPS layer and above, the manifestations of the vulnerabilities are diversified: some require the scripting capability of the browser while others use static HTML contents entirely; some require the HTTP-proxy mechanism enabled in the browser while others do not need this requirement. The existence of the vulnerabilities clearly undermines the end-to-end security guarantees of HTTPS.

People who are less familiar with HTTPS sometimes argue that the HTTPS security inherently depended on the trust on the proxy, and thus the assumption about a malicious proxy was inappropriate. This argument is conceptually incorrect since HTTPS' goal is to achieve the end-to-end security. Also, we show that in practice the trust on the proxy is too brittle for HTTPS to depend on. We constructed two versions of attack programs to show two levels of threats: (1) the first level, which is already serious, is due to the wide use of proxies for web access. The integrity of proxies is generally difficult to ensure. For instance, malware and attackers may take over legitimate proxies in many hotels and Internet cafes, because they are not

well managed. Many free third-party open proxies are also essentially unaccountable, etc; (2) the second level, which is more severe, is due to the fact that browsers' proxy-configuration mechanisms and browsers' communications with proxies are often unencrypted in many network environments. This makes a user vulnerable even when he/she is not knowingly connected to an untrusted proxy, as long as an attacker has the MAC layer access to the victim's network. In our Ethernet and WiFi experiments, the attacker simply needs to connect to the same Ethernet local area network (LAN) or wireless access point (AP) to launch the attacks. The damages of such attacks are the same as those caused by physically taking over a legitimate proxy. With the PBP vulnerabilities in browsers, the end-to-end security guarantees promised by HTTPS are lost because users basically need to trust the network in order to trust HTTPS.

5.2 Background

5.2.1 Same Origin Policy

Browsers support the functionality of downloading contents and executing scripts from different websites at the same time. Given some websites may contain malicious contents, it is crucial that browsers isolate the contents and scripts of different websites in order to prevent cross-domain interference. In addition, browser should allow scripts to access the contents of the same websites in order to perform normal web functionalities. This access-control policy is referred to as the same-origin policy.

Scripts and static contents are rendered and composed into webpages. The same-origin policy is enforced by isolating webpages according to their own security contexts derived from their URLs. A typical URL is represented in the format of `protocol://serverName:port/path?query` and the corresponding security context is a three-tuple <protocol,serverName,port>. As an example, the protocol can be HTTP or HTTPS, the serverName can be www.ebay.com, and the port can be 80, 443, or 8080, etc.

Each webpage is hosted in a frame or an inline frame. A browser window is a top level frame, which hosts the webpage downloaded from the URL shown in the address bar. A webpage can create multiple frames and inline frames to host webpages from different URLs. The access control mechanism between these webpages conforms to the same-origin policy described above. For example, suppose frame w1 loads a webpage from `https://bank.com` and frame w2 loads a webpage from `http://bank.com` or `https://evil.com`. If the script running in w2 attempts to access an HTML object inside w1, the access will be denied by the browser's security mechanism because of the same-origin policy. Without the same-origin policy, the document content of `https://bank.com` would be accessible to a script embedded in the webpage from `http://bank.com` (which could be faked by proxies and routers because it is not encrypted) or `fromhttps://evil.com`, which would defeat the purpose of HTTPS.

Similar to frame, other objects, such as XML and XMLHTTPRequest, rely on the same-origin policy to protect their documents as well. Also, webpages can be attached with a type of plain-text data called cookies. Cookies have a slightly different same-origin policy, which will be described in Section 5.4.2.

5.2.2 Basics of HTTPS and Tunneling

HTTPS is the protocol for HTTP communications over Secure Sockets Layer (SSL) or Transport Layer Security (TLS) [57]. For simplicity, in the rest of the paper, we use "SSL" to refer to both SSL and TLS. HTTPS is widely used to protect sensitive communications, such as online banking and online trading, from eavesdropping and man-in-the-middle attacks. At the beginning of an HTTPS connection, the browser and the web server go through an SSL handshake phase to ensure that: 1) the browser receives a legitimate certificate of the website issued by a trusted Certificate Authority (CA); and 2) the browser and the server agree on various cryptographic parameters, such as the cipher suite and the master key, in order to secure their connection. Once the handshake succeeds, encrypted data flow between the browser and the server. A

malicious proxy or router may disrupt the communication by dropping packets, but it should not be able to eavesdrop or forge data.

All major browsers support HTTPS communications through HTTP proxy. The mechanism is referred to as "tunneling". Before starting the SSL handshake, the browser sends an HTTP CONNECT request to the proxy, indicating the server name and port number. The proxy then maintains two TCP connections, with the browser and with the server, and serves as a forwarder of encrypted data. To tunnel the HTTPS packets between the two TCP connections, the proxy needs to set different values in the IP and TCP headers, such as IP addresses and port numbers. But it is not able to manipulate the encrypted payload besides copying it byte-by-byte. Therefore, the proxy does not have any additional information about HTTPS traffic beyond the IP and TCP headers. Normally an adversary must break the cryptographic schemes used by HTTPS in order to access the actual HTTPS contents. Note that a proxy is not a trusted entity in HTTPS communications. By design, confidentiality and authenticity of HTTPS should be guaranteed when the traffic is tunneled through an untrusted proxy; in reality, as we will show in Section 5.5, proxies are widely used in many network environments where proxies are not expected to be trustworthy. Being merely an interconnecting host on the network, the proxy is not a trusted entity that the HTTPS security relies on.

In the next two sections, we describe PBP attack scenarios. The versions of the browsers in our discussion are IE 7, IE 8, Firefox 2, Firefox 3, Opera 9, Chrome Beta and Chrome 1.

5.3 Script-Based PBP Exploits

Scripting is a critical capability of modern browsers. However, they impose more risks than static HTML contents if the scripting mechanism is not carefully designed and evaluated against different types of adversaries. Cross-site scripting [58] and browser cross-domain attacks [36] are the representative examples of vulnerabilities

exposed by scripting. While these attacks have provoked many discussions in the web security community, so far there has been less attention on the possibility of script-based attacks against HTTPS when the proxy is assumed the adversary.

In this section, we will describe several script-based attacks, some of which are because of executing regular HTTP scripts in the HTTPS context while others are because of executing scripts from unintended HTTPS websites in the context of target HTTPS websites. These attacks raise a concern that browsers' scripting mechanisms have not been thoroughly examined under the PBP adversary.

5.3.1 Embedding Scripts in Error Responses

We explained earlier that the browser sends an HTTP CONNECT request to the proxy when it tries to access an HTTPS server through the proxy. Sometimes the proxy may fail in connecting to the target server, in which case the proxy should send an HTTP error message back to the browser. For instance, when the browser requests `https://NonExistentServer.com`, the proxy will return an HTTP 502 Proxy Error message to the browser because the proxy cannot find a valid IP address associated with the server name NonExistentServer.com. Note that the communication between the browser and the proxy still uses plain-text up to this point. Interestingly enough, the browser renders the error response in the context of `https://NonExistentDomain.com`, although the server does not exist. We observed this behavior on all browsers that we studied. In addition to HTTP 502, other HTTP 4xx and 5xx messages are treated in a similar way.

The attack is illustrated in Figure 5.2. Since the browser completely relies on the proxy for the tunneling, the proxy can arbitrarily lie to the browser, which leads to the compromise of HTTPS confidentiality and authenticity. We now use an example to illustrate how a PBP adversary can steal the sensitive data from the browser when it is visiting an HTTPS server. Suppose the browser is accessing `https://myBank.com`, upon receiving the HTTP CONNECT request from the browser, the proxy

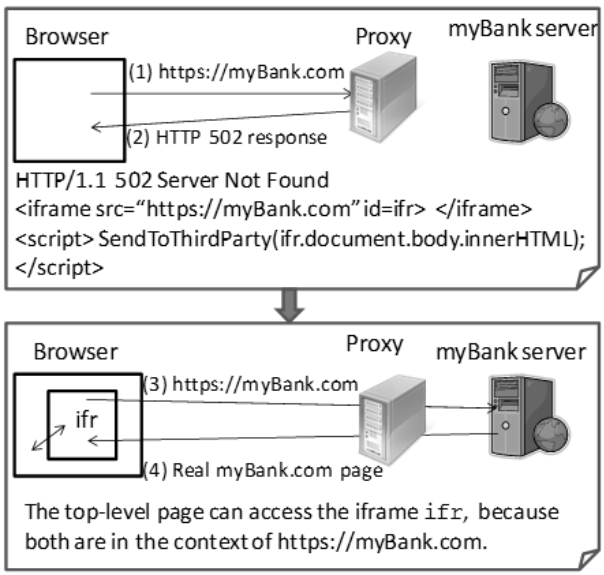

Figure 5.2. The attack embedding scripts in 4xx/5xx error messages

may pretend that the server did not exist by returning an HTTP 502 error message. The error message also includes an iframe (inline frame) and a script. When the browser renders the error message, the iframe will cause the browser to send another CONNECT request for `https://myBank.com`. The proxy will behave normally this time by tunneling the communication to the server. Thereafter, user's banking data will be loaded into the iframe (abbreviated as ifr). However, the script embedded in the original error message has been running in the context of `https://myBank.com`. This allows the script to reference ifr.document and send the user's banking data (e.g., body.innerHTML) to a third party machine, circumventing the same-origin policy of the browser. Besides peeking the user's banking data, the attacker can also transfer money from the bank on behalf of the user.

The attack does not depend on which authentication mechanism is used between the victim and the server. For instance, if the server uses password authentication, the proxy can behave benignly until the victim successfully logs on, and then launch the attack. The situation is much worse if the server uses Kerberos authentication

(similarly, NTLM authentication), in which case the authentication happens automatically without asking the user for the password. The attack can be launched even when the victim does not intend to visit the HTTPS server: whenever the victim visits a website `http://foo.com`, e.g., a popular search engine, the proxy may insert the following invisible iframe into the webpage of foo.com to initiate the same attack.

```
<iframe src="https://SiteUsingKerberos.com" style="display:none">
</iframe>
```

Kerberos is typically used in enterprise networks. This vulnerability allows the proxy to steal all sensitive information of the victim user stored on all HTTPS servers in the enterprise network, once the user visits an HTTP website.

5.3.2 Redirecting Script Requests to Malicious HTTPS Websites

After describing the PBP attacks based on the mishandling of HTTP 4xx and 5xx error messages in browsers, we now turn to another security flaw that can be exploited when browsers are dealing with HTTP 3xx redirection messages.

A benign redirection scenario is: when the user makes a request to `https://a.com`, the proxy can return a response, such as "302 Moved temporarily. Location: `https://b.com`", to redirect the browser to `https://b.com`. Similar to the previous scenario, the redirection message is in plain-text. The redirection is explicitly processed by the browser, so there is no confusion about the security context of the page – the page of the redirection target will be rendered in the context of `https://b.com`. In other words, a request redirected to `https://b.com` is equivalent to a direct request to `https://b.com`. There seems no security issue here.

However, the ability for a proxy to redirect HTTPS requests can be harmful when we consider the following scenario(see Figure 5.3): many webpages import scripts from different servers. For instance, a page of `https://myBank.com` may include a script `https://scriptDepot.myBank.com/foo.js` or a third-party script `https://x.akamai.net/foo.js`. According to the HTML specification, a script element

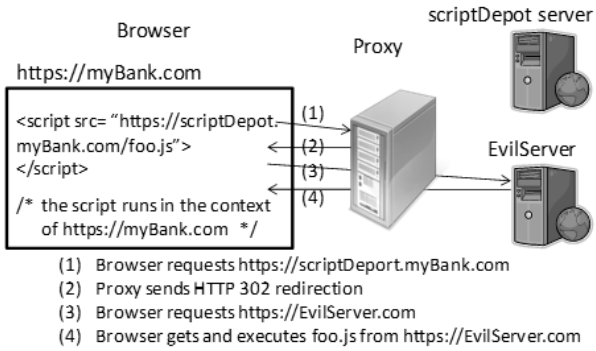

Figure 5.3. The attack using 3xx redirection message

does not have its own security context but instead runs in the context of the frame that imports it. To launch an attack, a proxy may simply use a 3xx message to redirect an HTTP CONNECT request for `https://scriptDepot.myBank.com` or `https://x.akamai.net` to `https://EvilServer.com`. This will cause the script `https://EvilServer.com/foo.js` to be imported and run in the context of `https://myBank.com`. Once the script runs, it can compromise the confidentiality and authenticity of the communication in a similar manner as described previously.

This attack affects Firefox and Opera. IE and Chrome are immune because they only process HTTP 3xx messages after the SSL handshake succeeds. In other words, 3xx messages from the proxy are ignored by the browser for HTTPS requests.

5.3.3 Importing Scripts Into HTTPS Contexts Through "HPIHSL" Pages

Many web servers provide services of HTTP and HTTPS simultaneously. Normally, sensitive webpages, such as user login, personal identification information, and official announcement, are accessible only via HTTPS to prevent information leak and forgery. Less critical webpages are accessible via HTTP for reduced processing overhead. Webpages often need to import additional resources, such as images, scripts, and cascade style sheets. When a page is intended for HTTP, the resources

are usually fetched using HTTP as well, because the page is not intended to be secure against the malicious network anyway.

However, the reality is that although less-sensitive webpages are intended to be accessed via HTTP, most of them actually can also be accessed via HTTPS. We refer to these pages as HTTP-Intended-but-HTTPS-Loadable pages, or "HPIHSL pages". When a HPIHSL page loaded in the HTTPS context imports resources using HTTP, browsers display different visual warnings: 1) IE pops up a yes/no dialog window. If user clicks no, the resources retrieved via HTTP will not be rendered and the lock icon will stay in the address bar. Otherwise, the resources will be rendered but the lock icon is removed; 2) Firefox pops up a warning window with an OK button. After user clicks it, the HTTP resources are rendered and a broken lock icon is displayed on the address bar. 3) Opera and Chrome automatically remove the lock icon (or replace it with an exclamation mark) to indicate that HTTP resources have been imported.

We found that the code logic for detecting HTTP contents in HTTPS pages is triggered only when the browser needs to determine whether to invalidate/remove the lock icon on the address bar, which is only correspondent to the top-level frame of the browser. Therefore, when the top-level frame is an HTTP page, the detection is bypassed even when this HTTP page contains an HTTPS iframe that loads an HPIHSL page.

This turns out to be a fatal vulnerability for many real websites. For example, a PBP can steal the user's login information from the HTTPS checkout page of j-Store.com (the first row of Table 5.1): when the user visits an HTTP merchandise page on j-Store.com, the proxy can insert the following invisible iframe into the page:

```
<iframe src="https://www.j-Store.com/men-shoes.html"
  style="display:none">
</iframe>
```

Without users' awareness, the invisible iframe loads the HPIHSL page men-shoes.html via HTTPS. Because this page requests a script from `http://switch.`

Table 5.1
HTTPS domains compromised because HPIHSL pages import HTTP scripts or style-sheets

Compromised HTTPS domain (the domain names are obfuscated)	The HPIHSL page that imports scripts or CSS	Domain and path of the HTTP script or CSS imported by the HPIHSL page
https://www.j-store.com The checkout service is in this domain	The "men's shoes" page in www.j-store.com	http://switch.atdmt.com/jaction/
https://www.OnlineServiceX.com The checkout service is in this domain	The account help page at www.OnlineServiceX.com/support /account	http://www.OnlineServiceX.com/ support/accounts/ bin/resource/
https://www.s-store.com The checkout service is in this domain	The "Appliances" page in www.s-store .com	http://content.s-store.com/js/
https://www.CertificateAuthorityX.com A leading certificate authority	The "repository" page in www. CertificateAuthorityX.com imports a CSS	http://www.CertificateAuthorityX .com/css/
https://www.eCommerceX.com The checkout and user profiles are in this domain	The homepage of www. eCommerceX.com	http://images.eCommerceX.com /media/
https://www.sb-store.com The checkout service is in this domain	The "Furniture" page in www.sb-store.com	http://graphics.sb-store.com /images/
https://www.CreditCardX.com A credit card company	The homepage of www.CreditCardX.com	http://switch.atdmt.com/jaction/ COF_Homepage/v3/
https://www.b-bank.com A bank in the Midwest	The page www.b-bank.com/ford.asp	http://www.google-analytics .com/
https://CodeRepositoryX.net Open source projects management system. User logins are in this domain.	The homepage of CodeRepositoryX.net	http://pagead2.googlesyndication .com/
https://uboc.MortgageCompanyX.com A California mortgage company	The homepage of uboc.MortgageCompanyX.com	http://uboc.MortgageCompanyX .com/Include/Utilities/ClientSide/
https://cs.University1.edu, the login system is in this domain	The homepage of cs.University1.edu	http://tags.University1.edu/
https://www.eecs.University2.edu	a student's homepage www.eecs. University2.edu/ ax	codice.shinystat.com/cgi-bin/

`atdmt.com/jaction/` via HTTP, the proxy can provide a malicious script to serve the request. Since the script is in the inserted iframe, it will run in the context of `https://www.j-Store.com`. The PBP also overwrites the "checkout" button on the HTTP merchandise page so that when the user clicks on it, the HTTPS checkout

page opens in a separate tab. The personal data entered by the user therefore can be easily obtained by the proxy's script in the invisible iframe. In addition, the proxy can impersonate the logon user to place arbitrary orders. We believe that this is a significant browser weakness: as long as any HPIHSL imports scripts or style-sheets (usually via HTTP as we explained), the entire HTTPS domain is compromised.

To get a sense about the pervasiveness of vulnerable websites, one of the authors of this paper used HTTPS to visit HPIHSL pages for a few hours. Table I shows twelve websites that we confirmed vulnerable (the exact names of the websites are obfuscated). Each row also shows the problematic HPIHSL page and the domain of the imported script. The vulnerable websites covered a wide range of services such as online shopping, banking, credit card, open source projects management, academic information, and certificate issuance. In particular, even the homepage domain of a leading certificate authority was affected. It is a reasonable concern that many websites simultaneously opening HTTP and HTTPS ports are vulnerable.

5.4 Static-HTML-Based PBP Exploits

We just described a number of script-based attacks that violate the same-origin policy. By running malicious scripts in the context of victim HTTPS domains, these attacks can access or alter sensitive data that are supposed to be protected by HTTPS.

Nevertheless, in order to better understand the potential threat of PBP, thinking beyond script-based attacks is very important. Typically, for script-based security issues, the defense solutions are along the line of disabling, filtering, or guarding scripts. When a class of security problems is not always script-related, defense solutions should be explored more broadly.

In this section, we show two attacks that can be accomplished entirely by static HTML contents. They target the authentication mechanisms in browsers. In the first attack, the proxy's own page can be certified with the trusted certificate of the

HTTPS server that the browser intends to communicate. In the second attack, the proxy can authenticate to the HTTPS server as a logon user.

5.4.1 Certifying a Proxy Page with a Real Certificate

In Section 5.3.1, we have seen that the PBP proxy can supply a script in an error-response. The script will run in the HTTPS context of the victim server and compromise the confidentiality. When we reported this issue to a browser vendor, one of the vendor's proposed fixes was to disable scripts in any 4xx/5xx error-response pages, and only render static HTML contents. The proposal was based on the consideration that benign proxy error messages are valuable for users to troubleshoot communication problems, but there is no compelling reason to allow scripts in error messages.

This fix would not block the attack that we describe below, which does not involve any script. Figure 5.4 illustrates how a proxy certifies a fake login page by taking advantage of a cached certificate of `https://www.paypal.com` from a previous SSL handshake. (Note that it is a browser bug. PayPal represents an arbitrary website.) IE, Opera and Chrome, but not Firefox, are vulnerable to this attack.

The attack works as follows: when a browser issues a request for `https://www.paypal.com` (step 1), the proxy returns an HTTP 502 message (or any other 4xx/5xx message) that contains a meta element and an img element (step 2). The meta element will redirect the browser to `https://www.paypal.com` after one second. But before the redirection, the following steps happen subsequently: the img element requests an image from `https://www.paypal.com/a.jpg` (step 3). In order to get a.jpg, the browser initiates an SSL handshake with the HTTPS server. The request is permitted by the proxy at this time. After the browser receives a legitimate certificate from the HTTPS server (step 4), it will try to retrieve a.jpg, which may or may not exist on the server (not shown in the figure). But its existence is not important here because the purpose of the img element is to acquire a legitimate certificate, which

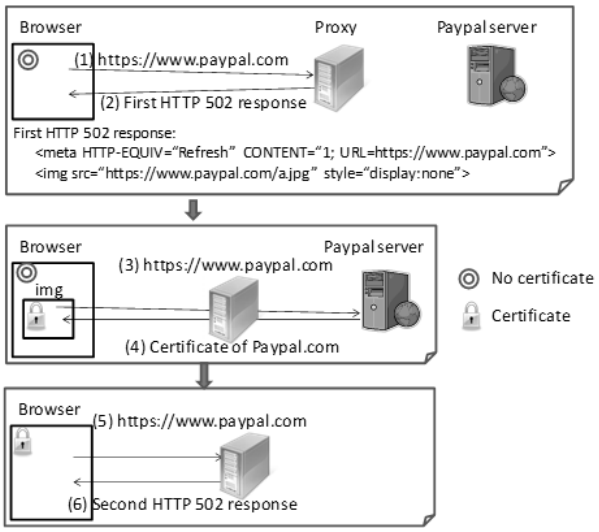

Figure 5.4. The attack certifies a faked login page as https://www.paypal.com

has been cached in the browser now. The certificate cache is designed to enhance the performance of HTTPS by avoiding repetitive re-validation for each SSL session.

When the one-second timer is expired, the browser will be redirected to `https://www.paypal.com` (step 5). This time, the proxy returns another HTTP 502 message (or any other 4xx/5xx message) that contains a fake login page (step 6). When the browser renders this page, it picks up the cached certificate of PayPal and displays it on the address bar as if the fake page was retrieved from the real `https://www.paypal.com`.

While the attack described here and the one described in Section 5.3.1 both take advantage of the fact that browsers render proxy's error messages in the context of HTTPS servers, these two attacks are distinguishable - In terms of the technique, this is a perfect GUI spoofing attack. Even when the user starts a fresh browser and uses a bookmark to access the HTTPS URL, he/she still gets the certified faked page. The attack is conducted in only one window and does not execute any script, therefore bypasses the pop-up blockers in today's browsers that will otherwise thwart the spoofing attack. No other attack that we describe can achieve the same result.

In terms of the root cause, the proxy-page-context problem in Section 5.3.1 alone does not necessarily enable this attack, e.g., we have confirmed that Firefox is not vulnerable to the attack although it has the problem in Section 5.3.1. A key enabler of the GUI spoofing attack is the interaction between the graphic interface and the certificate cache: for IE, Opera and Chrome, the certificate is displayed as long as it is available in the cache.

5.4.2 Stealing Authentication Cookies of HTTPS Websites by Faking HTTP Requests

The attack in Section 5.4.1 is to impersonate a legitimate HTTPS website. We now describe an attack that allows the PBP to impersonate victim users to access HTTPS servers by stealing their cookies.

Cookies are pieces of text that browsers receive from web servers and store locally. They are used to maintain the states of HTTP transactions, such as items in consumer's shopping carts and personalized settings of user webpages. In addition, they are used as an important mechanism for web servers to authenticate individual users. After a user successfully logs on a server, the server sends some cookies to be stored in the user's browser, which uniquely identify the session between the server and the user. Next time when the user accesses the server, these cookies are presented to the server as a proof of the identity of the user.

Browsers use the same-origin policy to determine whether cookies can be attached to requests or accessed by scripts. The policy specifies that: 1) Cookies of a domain can only be attached to the requests to the same domain; 2) Cookies of a domain are only accessible to scripts that run in the context of the same domain. However, unlike the same-origin policy of script and DOM, the same-origin policy of cookies does not make a distinction between HTTP and HTTPS by default. In the default scenario, cookies of `http://a.com` may be accessed by pages or scripts of `https://a.com`, and vice versa. Optionally, a SECURE attribute [59] can be set to ensure that cookies

Table 5.2
Insecure HTTPS websites due to the improper cookie protection

Website (names are obfuscated)	URL (obfuscated)	Description
StockTrader	https://trading.StockTrader.com	A leading stock brokerage company
eCommerce X	https://www.eCommerceX.com	A leading online store
V-Bank	https:// online.vbank.com	Online banking
Manuscript Manager	https://mc.ManuscriptManager.com	The submission and review system of an academic/engineering society
Travel Company	https://www.TravelCompany.com	A leading Internet travel company
GMail (not obfuscated as it is publicly known)	https://mail.google.com	Google's email service
Mortgage Company Y	https://MortgageCompanyY.com	Mortgage lender
Utility Company X	https://www.UtilityCompanyX.com	A utility company in the west coast of the United States
Goverment Service X	https://egov.GovermentServiceX.gov	A web service for U.S. immigration cases

can only be read by pages in the HTTPS context and be attached to the HTTPS requests (of course, after the SSL handshake).

We found that many websites do not set the SECURE attribute for cookies that identify HTTPS sessions . As an example, an author of the paper investigated about 30 websites in which he owns an account. About one-third of the websites used cookies for authentication but did not set the SECURE attribute for them. Every website was verified individually to show that the stolen cookie was sufficient to allow the attacker to get into the logon session from an arbitrary machine and to perform arbitrary operations on behalf of the victim user. These nine websites are listed in Table 5.2, with their names and URLs obfuscated. They cover a wide range of services such as stock broker, online shopping, online banking, academic paper reviewing, email service, mortgage payment, utility billing, government service, and traveling. They affect many different aspects of a person's online security.

It is straightforward to launch the attack: the proxy waits until the user logs into the server (usually after seeing a few CONNECT requests), e.g., the stock trading website `https://trading.StockTrader.com`. After that, once the browser re-

quests any HTTP page (including a page requested from another browser tab or any tool bar), the proxy embeds an iframe of `http://trading.StockTrader.com` in the HTTP response. When the browser renders the iframe, it makes an HTTP request for `http://trading.StockTrader.com`, exposing the authentication cookie in plain text to the proxy.

Given that a significant fraction (one-third) of the HTTPS websites that we examined have this problem and many of them are reputable, we believe this vulnerability exists in many other HTTPS websites as well. Although it is possible that inexperienced developers do not have knowledge about the SECURE attribute of cookies, the fact that reputable websites also make this mistake suggests that the concept of the SECURE attribute is commonly misconceived. The SECURE attribute is often vaguely defined as a mechanism to prevent malicious HTTP pages. It is never made clear that when the network is assumed untrusted, the SECURE attribute should be considered as a mechanism to prevent malicious proxies and routers. Without this clear interpretation, a developer might have a misconception: my HTTP pages are very secure (or "my website does not run HTTP at all"). Why bother to prevent my own HTTP pages from stealing cookies of the HTTPS sessions on my website?

5.5 Feasibility of Exploitation in Real-World Network Environments

By definition, the security of communications over HTTPS should not rely on the integrity of any intermediate node in network path, such as proxies and routers. As described in the previous sections, however, the guarantees of HTTPS can be subverted when a malicious or compromised proxy is being used. There are many circumstances where proxies are commonly used and therefore the PBP vulnerabilities can be easily exploited: (1) Mobile environments such as conference rooms, airports, hotels and hospitals [60]; (2) Corporate and university networks, e.g., Microsoft's corporate network and the campus networks in Berkeley and UCSD [61]; (3) Free third-party proxies on the Internet [62]. In these cases, proxies may be used for

various legitimate reasons, such as billing, traffic regulation, and traffic anonymization. However, if they are infected by viruses, hijacked by attackers, or configured by malicious insiders, the PBP attacks can be launched.

In this section, we will show that in real-world network environments, the PBP vulnerabilities can be exploited more easily than hacking into the proxy machine. An attacker can exploit the vulnerabilities even when the victim is not knowingly using an untrusted proxy. The attacker only needs the capability of sniffing users' traffic and sending fake packets back to browsers. An attacker can easily do this by sitting in the vicinity of victim users in a wireless environment or connecting to the same local area network (LAN) of victim users in a wired environment. Note that the attack scenarios to be described do not show any additional vulnerability, but demonstrate that the PBP vulnerabilities described earlier result in serious consequences for people's online security.

The tactic of our attacks is to impersonate a legitimate proxy or insert an unwanted proxy into the communication path without the user's awareness. We will discuss a few basic elements in this tactic: (1) TCP hijacking - It is a known fact that anyone who can sniff IP packets can hijack the TCP connections, and thus impersonate clients and servers; (2) Proxy-Auto-Config (PAC) mechanism [63] - Alternative to manual configuration, browsers use the PAC mechanism to obtain a script from a server and configure proxy settings by the script; (3) Web-Proxy-Auto-Discovery protocol (WPAD) [64] - All browsers support WPAD. WPAD makes the proxy configuration completely under the hood: it attempts to discover a proxy, and automatically falls back to the "no-proxy" setting if the attempt fails. Using WPAD, the same browser machine can access the web at office, hotel and home without changing any setting. Since TCP, PAC and WPAD are not cryptographic protocols, they are not expected to be resilient against an attacker who can access the network traffic. However, combining these facts with PBP vulnerabilities, HTTPS' properties become very easy to break in reality.

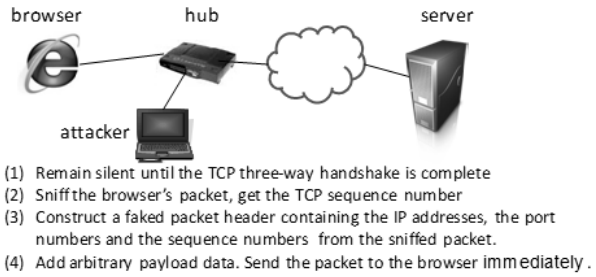

Figure 5.5. A typical TCP hijacking

We have built Ethernet and wireless testbeds to show various attack scenarios. The details are provided in the following subsections.

5.5.1 A Short Tutorial of TCP Hijacking

It is well known that an attacker who can sniff TCP traffic can impersonate the sender or the receiver of a TCP connection. This technique is referred to as TCP hijacking and is shown in Figure 5.5. The attacker is at a location where he can sniff the TCP packets between the browser machine and the server. To simplify description, we assume that the attacker connects to the same Ethernet hub as the user machine. When the browser tries to establish a TCP connection with the server, the attacker does nothing but wait for the completion of TCP three-way handshake. When the attacker receives the packet which contains an HTTP request sent by the browser, it parses the packet to extract the IP addresses, the port numbers, and the current sequence numbers of both the browser and the server. It then uses this information to fake a server response packet with appropriate IP and TCP headers and payload data. The fake packet is immediately sent back to the browser.

Since the attacker can only sniff but not intercept packets, the server will still return a legitimate response packet to the browser. Given that the distance between the attacker and the browser is equal or shorter than the distance between the browser and the server, the fake response packet generated by the attacker almost always

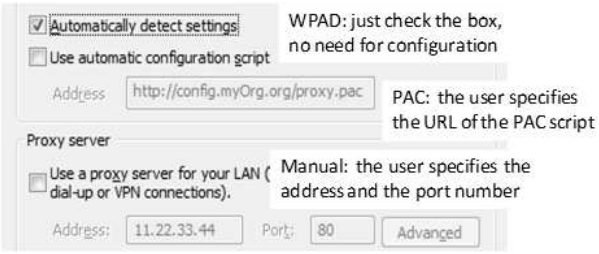

Figure 5.6. Proxy setting options for IE and Chrome

arrives before the legitimate response packet from the server. Although this is a race between the attacker and the server, because the attacker has already prepared most of the faked response and only waits to fill in a few header fields, it is easy to win the race. The browser will accept the fake packet and discard the legitimate packet as a duplicate, because both packets have the same TCP sequence number.

5.5.2 PBP Exploits by a Sniffing Machine

As we stated earlier, connecting to a proxy is necessary in many circumstances, such as corporate networks, hotels, and conferences, for the purpose of billing or auditing [60–62]. Browsers' proxy settings can be configured manually, or by specifying the URL of the PAC script, or by WPAD. The attacker has several options accordingly. Figure 5.6 shows the user interface of proxy settings for IE and Chrome. Other browsers' user interfaces are almost functionally identical.

Browsers with manual proxy-settings. Manual configuration requires the (advanced) user to enter the hostname/IP address and the port number of the specific proxy server. The attacker needs to hijack the TCP connection between the browser and the proxy to impersonate the proxy. Browsers configured by PAC scripts [63]. A browser can fetch a PAC (Proxy Auto-Config) script from a server by specifying the URL, such as `http://config.myOrg.org/proxy.pac`. The script contains a special function FindProxyForURL(url,host), which returns a string containing one or more

proxy specifications given a URL and a hostname. In practice, proxy settings are normally cached for better performance. To attack this browser, the attacker can hijack the TCP connection to impersonate the PAC server config.myOrg.org. The following PAC script is served to the browser. The browser will use"`proxy.evil.com:80`" as its proxy.

```
function FindProxyForURL(url,host) return "PROXY proxy.evil.com:80";
```

The advantage of impersonating the PAC server, compared to impersonating the proxy server, is that the hijacking only needs to be done once and the browser's proxy setting will be changed permanently.

Browsers enabling WPAD [64]. WPAD (Web Proxy Auto Discovery) is the only option for users to browse the web from different networks without changing the configuration. When WPAD is enabled, the browser does not initially know the URL of the PAC script, but asks the DHCP server for it. If DHCP server does not have the information or does not respond, the browser asks the DNS servers. Once the URL of the PAC script is obtained, the browser fetches the script and configures its proxy settings. If the browser cannot find any proxy configuration script, it automatically falls back to the "no-proxy" state, in which the browser does not access the web through any proxy. Our attack program sniffs browser packets. When there is a WPAD query for DHCP or DNS server, the program replies immediately with the URL of a malicious PAC script on the attacker machine.

Home networks typically have no HTTP proxy servers, so it may be an expectation that online banking at home is secure. It is worth noting that whether there is a proxy in a home network does not affect the security. The security is only affected by whether the browser has any one of the proxy settings enabled. For example, if a laptop sets the WPAD capability in the office hours in a corporate network, it will be insecure to do online banking at home in the evening with the proxy setting unchanged, because the attacker can fake a WPAD response to convince the browser that there is a proxy.

If a user does disable proxy service in browsers, the vulnerabilities described in Section 5.3.1, 5.3.2 and 5.4.1 are no longer exploitable because the browser will directly establish HTTPS connections with servers instead of tunneling the connections through proxies, evading the code paths that trigger the vulnerabilities. However, the remaining two vulnerabilities described in Section 5.3.3 and 5.4.2 can be exploited as they only require attackers to sniff HTTP requests and forge HTTP responses.

The default proxy settings of the browsers. If a user has never modified any proxy settings since the installation of the browser, the default settings vary in different browsers: (1) Firefox does not enable any proxy setting by default; (2) IE enables and uses WPAD in its very first run after the installation. If this first use is successful, the WPAD setting is checked, otherwise it is unchecked. Chrome always uses IE's setting; (3) After the installation, Opera's initial setting is the same as IE's setting. Therefore, even in a fresh IE, Opera or Chrome at home, the proxy setting will be enabled if the attacker responds to all WPAD requests that he/she receives.

5.5.3 Attack Implementations

We implemented all attack scenarios in both the Ethernet and wireless environment. In the Ethernet, we used WinPcap [65] to sniff and inject packets in Windows platform. WinPcap is a network monitoring library; it provides a set of APIs which allow us to capture all raw packets received by network interface card (NIC) and send raw packets through NIC. These raw packets include link layer headers, IP headers, TCP headers, and full payload data. While a NIC normally discards packets whose physical (MAC) addresses do not match that of the NIC, WinPcap can set the NIC in the promiscuous mode such that all packets received by the NIC will be passed up.

Wireless environments are more dangerous. Given the nature of wireless networks, attackers can sniff wireless packets in the air when they are in the vicinity of the wireless access points which victims are using (unless per-user encryption schemes WPA and WPA2 are deployed, which will be discussed in Section 5.6). Conceptually,

the attacks in a wireless network are the same as that in an Ethernet LAN. However, we need to resolve a number of implementation issues to enable the attacks, which are described below.

Although WinPcap works well on most Ethernet NICs, it is not properly supported by many wireless NICs. First, many wireless NICs do not support the promiscuous mode for power conservation. Second, WinPcap device driver assumes Ethernet as its default link layer, which is incompatible with most wireless NICs. However, we do find that certain wireless NICs (e.g. Dell TrueMobile 1300 WLAN Mini-PCI Card) work with WinPcap and support the promiscuous mode. On these NICs, WinPcap emulates an Ethernet layer by automatically creating fake Ethernet frames from WiFi frames. In addition, we have developed a specific packet sniffer/injector that works with D-Link AG-132 Wireless USB Adapter in Windows platform.

5.6 Mitigations and Fixes

In this section, we describe how browsers vendors fixed or planned to fix the vulnerabilities reported in this paper. We also discuss possible ways to mitigate the impact of the class of PBP vulnerabilities before they are discovered and patched.

5.6.1 Fixes of the Vulnerabilities

We have reported these vulnerabilities (except the authentication cookie vulnerability) to the affected browser vendors: Microsoft's IE team, Mozilla's Firefox team, Opera Software and Google's Chrome team. Since the authentication cookie vulnerability in Section 5.4.2 is due to improper setting of cookie attribute by individual websites, we have to inform the websites instead of the browser vendors. Table 5.3 shows the browser vendors' responses to each of these vulnerabilities. The vendors have acknowledged the vulnerabilities reported by us. Although Mozilla has not explicitly confirmed a plan for fixing the vulnerabilities described in Section 5.3.1 and

Table 5.3
Vulnerability reporting and browser vendors' responses.

	Microsoft (IE)	Mozilla (Firefox)	Opera	Google (Chrome)
Vulnerability in Section 5.3.1	Fixed in IE8	Vulnerability acknowledged	Fixed in Dec.2007	Fix being developed
Vulnerability in Section 5.3.2	N/A	Vulnerability acknowledged	Fixed in Dec.2007	N/A
Vulnerability in Section 5.3.3	Vulnerability acknowledged Fix proposed	Vulnerability acknowledged Fix proposed	Vulnerability acknowledged	Fix planned
Vulnerability in Section 5.4.1	Fixed in IE8	N/A	N/A	Fix being developed
Vulnerability in Section 5.4.2	Not reported	Not reported	Not reported	Not reported

5.3.2, we believe that they will be fixed given the high risks. Microsoft has fixed them in IE8, and scheduled to patch IE7 soon. Opera has fixed them in December 2007. Google's fix is being developed.

IE8 and Opera fixed the vulnerability in Section 5.3.1 by displaying a local error page when receiving a 4xx/5xx response before the SSL handshake succeeds. Opera fixed the vulnerability in Section 5.3.2 by ignoring the proxy's 3xx redirections. As a proposal for the vulnerability in Section 5.3.3, Mozilla plans to fix it by blocking any script/CSS resources imported by HTTP into HTTPS context, or reliably display a warning. Microsoft and Opera are considering a "defense-in-depth" patch, of which the details have not been confirmed.

Browser vendors are not in the best position to fix the authentication cookie vulnerability described in Section 5.4.2, because currently there is no mechanism to for the browsers to know if a cookie value is for the authentication purpose or meant to be shared with the corresponding HTTP domain. The cookie's secure attribute largely depend on the application semantics.

5.6.2 Mitigations by Securing the Network

Because HTTPS is designed fundamentally for secure communications over an insecure network, it is of course an unconvincing "solution" that we secure the network in order to secure HTTPS. However, in practice, network-based mitigation approaches are still valuable to consider because it is not safe to assume every machine will be fully patched. More importantly, we believe there will be future vulnerabilities similar to what we have discovered. Good mitigations that are effective against known attacks may mitigate future/unknown attacks.

At the high level, users should be cautious about plugging their machines into untrusted network ports, connecting to unknown wireless access points (APs), or using arbitrary network proxies. In cases when users must go through them to access the network, they should avoid using the network for critical transactions, such as online banking. In enterprise networks where Kerberos authentication is being used, network administrators should prevent any unauthorized sniffing of user traffic.

Since the PBP attacks are so easy to launch if the attacker has the ability to intercept or sniff traffic content, a straightforward mitigation approach is to encrypt the content transmitted on the network. Fortunately, there have already existing techniques that are applicable in different scenarios:

- Almost all wireless APs support encryption, which make it difficult for adversaries to sniff traffic in the air. Among the commonly available encryption schemes, WPA and WPA2 are more secure, because they maintain per-user keys. WEP uses a static shared key among all the users who connect to the same AP. It is widely known that WEP is easy to break [66].

- Sometimes, users must rely on untrusted networks to access the Internet, e.g., in hotels, airports, conferences, and coffee shops. They may secure their traffic by using secure Virtual Private Network (VPN) if such option is available. Secure VPN allows a client to establish a secure connection with a VPN server, in which case all the traffic between them is encrypted. Once the connection is

established, all requests and replies will be tunneled through the VPN server. Conceptually the users' machines are connected to the enterprise network of the VPN server.

- Enterprise networks should deploy IPSec to encrypt traffic at the IP-layer. Today, IPSec coexists with regular IP in enterprise networks. There are lots of opportunities for PBP attackers to intercept or sniff users' traffic in enterprise networks. For example, large enterprise networks typically have thousands of network ports. Attackers can easily plug in their own wireless APs to a network port without being detected for a long time. These APs are often referred to as Rogue APs and allow attackers to gain unauthorized access to enterprise network (e.g., sniffing users' traffic). Another example is Network Load Balancing (NLB) where servers in the same load balancing group share a broadcast address [67]. This facilitates packet sniffing. To resolve these issues, it is important that all hosts involved in the communication must use IPSec to protect users from PBP attacks. To understand its importance, let us assume that IPSec is only used by all the proxy servers but not the PAC servers. Adversaries may still intercept/sniff the requests to PAC servers and feed malicious PAC scripts to browsers as we described earlier. Similarly, IPSec must be deployed on other types of servers that provide basic network services, such as DHCP servers and DNS servers.

6 DEFEATING CROSS-SITE REQUEST FORGERY ATTACKS WITH BROWSER-ENFORCED AUTHENTICITY PROTECTION

6.1 Motivation

Cross-site request forgery, also known as one-click attack or session riding and abbreviated as CSRF or XSRF, is an attack against web applications [5–7]. In a CSRF attack, a malicious web page instructs a victim user's browser to send a request to a target website. If the victim user is currently logged into the target website, the browser will append authentication tokens such as cookies to the request, authenticating the malicious request as if it is issued by the user.

A CSRF attack does not exploit any browser vulnerability. As long as a user is logged into the vulnerable web site, simply browsing a malicious web page can lead to unintended operations performed on the vulnerable web site. Launching such CSRF attacks is possible in practice because many users browse multiple sites in parallel, and users often do not explicitly log out when they finish using a web site. A CSRF attack can also be carried out without a user visiting a malicious webpage. In a recent CSRF attack against residential ADSL routers in Mexico, an e-mail with a malicious IMG tag was sent to victims. By viewing the email message, the user initiated an HTTP request, which sent a router command to change the DNS entry of a leading Mexican bank, making any subsequent access by a user to the bank go through the attacker's server [68].

CSRF appeared in the Open Web Application Security Project (OWASP) top 10 web application threats in 2007 (ranked at 5) [69]. Several CSRF vulnerabilities against real-world web applications have been discovered [70–72]. In 2007, a serious CSRF vulnerability in Gmail was reported [73]. It allowed a malicious website to surreptitiously add a filter to a victim user's Gmail account that forwards emails to a

third party address. CSRF vulnerabilities are very common. The potential damage of CSRF attacks, however, has not been fully realized yet. We quote the following from an online article [74],

> Security researchers say it's only a matter of time before someone awakens the "sleeping giant" and does some major damage with it – like wiping out a user's bank account or booking a flight on behalf of a user without his knowledge.
>
> "There are simply too many [CSRF-vulnerable Websites] to count," says rsnake, founder of ha.ckers.org. "The sites that are more likely to be attacked are community websites or sites that have high dollar value accounts associated with them – banks, bill pay services, etc."

Several defense mechanisms have been proposed and used for CSRF attacks. However, they suffer from various limitations (see Section 2.3).

We study browser-based defense against CSRF attacks, which is orthogonal to server-side defenses. The websites should follow the best practice to defend against the CSRF attacks before browser-side defenses are universally adopted. One crucial advantage of a browser-based solution compared with a server-side solution is that a user who started using the protected browser will immediately have all his web browsing protected, even when visiting websites that have CSRF vulnerabilities. Furthermore, because the number of major browsers is small, deploying protection at the browser end can be achieved more easily, compared with deploying server-side defenses at all websites.

We recognize that CSRF attacks are an example of the confused deputy problem. The current web design assumes that the browser is the deputy of the user and that any HTTP request sent by the browser reflects the user's intention. This assumption is not true as many HTTP requests are under the control of the web pages and do not necessarily reflect the user's intention. This becomes a security concern for HTTP requests that have sensitive consequences (such as financial consequences).

Our solution to this problem is to enhance web browsers with a mechanism ensuring that *all sensitive requests sent by the browser should reflect the user's intention.* We achieve that by inferring whether an HTTP request reflects the intention of the user and whether an authentication token is sensitive, and striping all sensitive authentication tokens from the HTTP requests that may not reflect the user's intention. We call it *Browser-Enforced Authenticity Protection.*

6.2 Understanding CSRF Attacks and Existing Defenses

CSRF attacks exploit existing authenticated sessions. Two common approaches for maintaining authenticated web sessions are cookies and HTTP authentication credentials, which we call *authentication tokens.*

Cookies [75] are pieces of text data sent by the web server to the browser. The browser stores the cookies locally and sends them along with every further request to the original web site who sets them. After a web site has authenticated a user, for example, by validating the user name and password entered by the user, the web site can send back a cookie containing a "session ID" that uniquely identifies the session, which is referred to as *authentication cookie.* If the web server relies only on cookies for user authentication, every request that has a valid authentication cookie is interpreted as an intended request issued by the authenticated user who owns the session. When sending a cookie to a browser, the website can specify an optional attribute expires among other three attributes. The expires field takes the value of a date that indicates how long the cookie is valid. After the date passes, the browser deletes the cookie. If the expires field is omitted, then the cookie is called a *session cookie* and should be deleted when user closes the web browser. Cookies with an expires field are called *persistent cookies.* Most financial websites and sensitive services specify the authentication cookie as a session cookie, because the session cookies are removed when the browser is closed and won't be abused by others who may share the same computer and browser.

HTTP authentication [76], an authentication mechanism defined in the HTTP protocol [77], is widely used within Intranet environments. In the mechanism, when accessing a webpage that requires authentication, the browser will popup a dialog asking for the username and password. After entering the information, the credential is encoded and sent to the web server via the Authorization request header. The browser remembers the credential until the browser is closed. When later the user visiting the webpages in the same authentication realm, the browser automatically includes the credential in the request via the Authorization header.

CSRF attacks use HTTP requests that have lasting observable effects at the web site. Two request methods are used in real-world HTTP requests: GET and POST. According to the HTTP/1.1 RFC document [77], the GET method, which is known as a "safe" method, is used to retrieve objects. The GET requests should not have any lasting observable effect (e.g., modification of a database). The operations that have lasting observable effects should be requested using the method POST. The POST requests have a request body and are typically used to submit forms. However, there exist web applications that do not follow the standard and use GET for requests that have lasting side effects.

Visiting web pages in one site may result in HTTP requests to another site; these are called cross-site requests. More precisely, in a cross-site request, the link of the request is provided by a website that is different from the destination website of the request. Cross-site requests are common. For example, a webpage may include images, scripts, style files and sub-frames from a third-party website. When the user clicks a hyper-link or a button contained in a webpage, the linked URL may be addressing a third-party website.

6.2.1 The CSRF Attack

The general class of cross site request forgery (CSRF) attacks was first introduced in a posting to the BugTraq mailing list [5], and has been discussed by web application

developers [6, 7]. CSFR attacks use cross-site requests for malicious purposes. For example, suppose that the online banking application of `bank.com` provides a "pay bills" service using an HTML form. The user asks the bank to send a check to a payee by completing the form and clicking the "Sumbit" button. Upon the user clicking the button, a POST request is sent to the server, together with the authentication cookie. When the web server receives this HTTP request, it processes the request and sends a check to the payee identified in the request.

A CSRF attack works as follows.While accessing the bank account, the user simultaneously browses some other web sites. One of these sites, evil.org, contains a hidden form and a piece of JavaScript. As soon as the user visits the web page, the browser silently submit the hidden form to `bank.com`. The format and content of the request is exactly the same as the request triggered by the user clicking the submit button in the "pay bill" form provided by the bank. On sending the request, the user's browser automatically attaches the authentication cookies to the request. Since the session is still active in the server, the request will be processed by the server as issued by user. As illustrated in this example, POST requests can be forged by a hidden form. If the bank uses GET request for the pay bill service, the request can be easily forged by using various HTML elements, such as ⟨img⟩, ⟨script⟩, ⟨iframe⟩, ⟨a⟩ (hyper-link) and so on.

We note that as long as a user is logged in to a vulnerable web site, a single mouse click or just browsing a page under the attacker's control can easily lead to unintended operations performed on the vulnerable web site.

CSRF vs. XSS. CSRF vulnerabilities should not be confused with XSS vulnerabilities. In XSS exploits, an attacker injects malicious scripts into an HTML document hosted by the victim web site, typically through submitting text embedded with code which is to be displayed on the page, such as a blog post. Most XSS attacks are due to vulnerabilities in web applications which fail in sanitizing untrustworthy inputs which might in turn be displayed to users. CSRF attacks do not rely on the execution and injection of malicious JavaScript code. CSRF vulnerabilities are due to the use of

Table 6.1
The CSRF vulnerabilities discovered in real world websites.

Vulnerable web site	**Targeted sensitive operation**
A university credit union site	Money transfer between accounts; adding a new account
A university web mail	Deleting all emails in the Inbox
An online forum for HTML development	Posting a message; updating user profile
Department portal site	Editing biography information

cookies or HTTP authentication as the authentication mechanism. A web site that does not have XSS vulnerabilities may contain CSRF vulnerabilities.

6.2.2 Real-world CSRF vulnerabilities

In order to understand how commonly the CSRF vulnerability exists in the real-world web applications, one of the authors of the paper examined about a dozen web sites for which he has an account and usually visits. As a result, we found four of them are vulnerable to CSRF attacks as shown in Table 6.1. We verified all the attacks with Firefox 2.0.

The university credit union site relies on session cookies for authentication. Some services provided in the online banking are vulnerable to the CSRF attack. In particular, adding new accounts and transferring money between accounts are vulnerable. In the experiment, we conducted a benign attack that transfers $0.01 from the victim's checking account to the saving account. We also successfully launch an attack to add an external account. Combining these two enables the adversary to transfer money from the victim's account to an arbitrary external account. Fortunately, the bank requires contacting the help-desk personally to confirm the operation of adding an external account. And also the bill paying service is not vulnerable.

The university web mail uses session cookies for authentication. Most sensitive operations (e.g., sending an email, changing the password) are protected against the

CSRF attacks using secret token validation (see Section 2.3). However, the feature of "managing folders" is vulnerable, and a CSRF attack can be launched to remove all emails in the victim's Inbox.

In an online forum for HTML development, all operations are vulnerable to the CSRF attack. The attacker is able to impersonate the victim user to send a posting, update the user profile, and so on. The vulnerable forum is created using phpBB [78], which is the most widely used open source forum solution. All forums created using phpBB 2.0.21 or earlier are vulnerable to the CSRF attack [79]. This is a well-known vulnerability and there are CSRF attack generators for phpBB forums available online. Many public forums have upgraded to phpBB 2.0.22 or later, but there are still many forums using the vulnerable versions.

In the departmental portal site, a CSRF attack is able to edit the biography information of the victim shown on the webpage.

We have reported the vulnerabilities to the websites of the university credit union and the university web mail; we did not expose the name of those websites here because they have not fixed the vulnerabilities yet. These examples of vulnerabilities demonstrate that there exist a considerable amount of web services vulnerable to the CSRF attacks and the potential damage could be severe.

6.2.3 A variant of CSRF attack

All existing CSRF defenses 2.3 fail when facing a variant of CSRF attacks mentioned in [80] and [47]. We use the Facebook as an example to illustrate the attack. Facebook allows the users to post an article or a video from any website to the user's own profile. For example, the user can post a video from `Youtube.com` to his Facebook profile by clicking "Share – Facebook" under the video. When clicking the link, the following GET request is sent to the Facebook: `http://www.facebook.com/sharer.php?u=http://www.youtube.com/watch?v=VIDEO_ID&t=VIDEO_TITLE`. This request loads a confirmation page (Fig. 6.1(A)) which asks the user the click a "Post" button

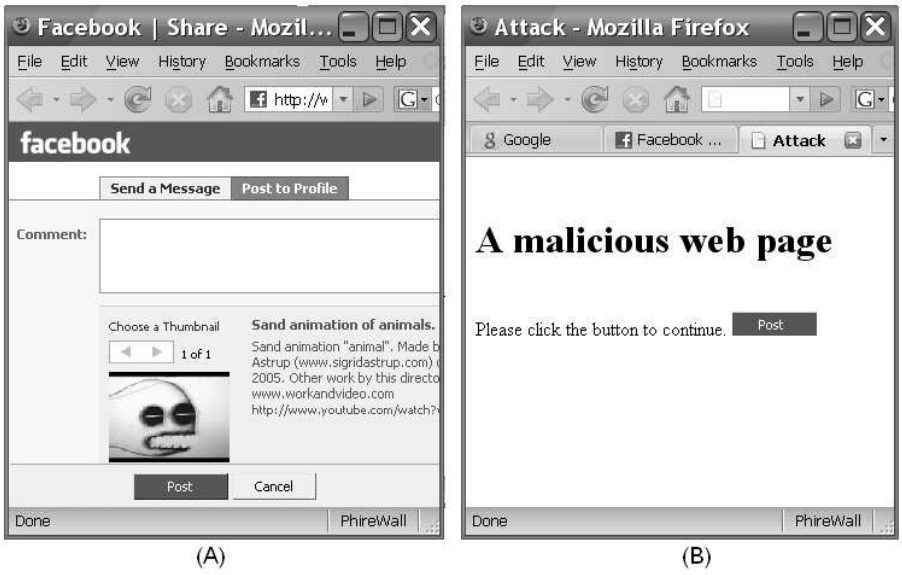

Figure 6.1. (A): The confirmation page that posts a video from Youtube.com to the Facebook profile; (B): A malicious page that includes (A) as an iframe and tries to trick the user click the button without seeing other parts of (A);

to complete the transaction. After the user clicking the "Post" button, a POST request is sent to `http://www.facebook.com/ajax/share.php` to confirm the posting operation.

An attacker is able to launch a CSRF attack that posts anything to the victim user's profile. On the malicious webpage, the attacker includes an iframe linking to the posting confirmation page (Fig. 6.1(A)). In addition, the attacker is able to auto-scroll the iframe to the "Post" button and hide other parts of the page by using two nested iframes and manipulating the sizes of the iframes. As a result, what is shown in the browser looks like Fig. 6.1(B). The user can be easily tricked to click the "Post" button without knowing that he is posting something to his own Facebook profile.

Facebook.com uses secret validation token to defend against CSRF attacks. However, because the request is sent by user clicking the "Post" button in the confirmation page provided by Facebook the request will include a correct validation token. Using

a referer-checking would also fail because the final posting request has a Referer header of Facebook.com.

This attack is traditionally defended using "frame busting", in which the target webpage includes a piece of JavaScript to force itself to be displayed in a top-level frame [81]. However, this defense can be defeated if the attacker disables the JavaScript in the sub-frame that links to the target webpage [82].

6.3 Browser-Enforced Authenticity Protection (BEAP)

CSRF attacks are particularly difficult to defend because cross-site requests are a feature of the web. Many web sites use legitimate cross-site requests, and some of these usages require the attachment of cookies to cross-site requests to work properly (e.g., posting a video from Youtube to Facebook in the above example). To effectively defend against CSRF attacks, one needs as much information about an HTTP request as possible, in particular, how the request is triggered and crafted. Such information is available only within the browser. Existing defenses suffer from the fact that they do not have enough information about HTTP requests. They either have to change the web application to enhance the information they have or to use unreliable source of information (such as Referer header). Even when such information is available, it is still insufficient. For example, they cannot defend against the attack in Section 6.2.3 because while they can tell the request is coming from their web form, they do not know that the web form is actually embedded in a page controlled by the attacker.

We focus on browser-based defense against CSRF attacks. It is well known that CSRF is a confused deputy attack against the browser. The current web design assumes that the browser is *always* the deputy of the user and that any HTTP request sent by the browser reflects the user's intention. This assumption is not true as many HTTP requests are under the control of the web pages and do not necessarily reflect the user's intention. This confusion causes no harm when these requests have no sensitive consequences, and merely retrieve web pages from the web

server. However, when these requests have sensitive consequences (such as financial consequences), it becomes a severe security concern. Because such requests occur in authenticated sessions, these requests have authentication tokens attached. The fundamental nature of the CSRF attack is that the user's browser is easily tricked into sending a sensitive request that does not reflect the user's intention.

Our solution to this problem is to directly address the confused deputy problem of the browser. More specifically, we propose Browser-Enforced Authenticity Protection (BEAP), which enhances web browsers with a mechanism ensuring that *all sensitive requests sent by the browser reflect the user's intention.* BEAP achieves this through the following. First, BEAP infers whether an HTTP request reflects the intention of the user. Second, BEAP infers whether authentication tokens associated with the HTTP request are sensitive. An authentication token is sensitive if attaching the token to the HTTP request could have sensitive consequences. Third, if BEAP concludes that an HTTP request reflects the user's intention, the request is allowed to be sent with authentication tokens attached. If BEAP concludes that an HTTP request may not reflects the user's intention, it strips all sensitive authentication tokens from the HTTP request. In this rest of this section, we describe BEAP in details.

6.3.1 Inferring the User's Intention

In inferring whether an HTTP request reflects the user's intention, we classify the requests into two types depending on the source of the request. Type-1 requests are caused by the webpages hosted in the browser. When displaying a webpage, the browser may send additional requests to retrieve the resources included in the web page, such as images, scripts and so on. These resources may come from the same website or a third-party website. Similarly, when the user clicks a hyper-link or a button contained in a webpage, requests are sent by the browser. In addition, the Javascripts contained in the webpages may send requests as well. In all these

cases, the URLs and contents of the requests are determined by the source webpage. Whether such a request reflects the user's intention is inferred by *browser-enforced Source-set checking*, which we will explain soon.

Type-2 requests are not associated with a source webpage. For example, when the user clicks an URL embedded in an email, the URL is passed to the browser as a startup argument, resulting in an HTTP request that is not associated with any webpage already hosted in the browser. We use the following *user-interface intention heuristics* to infer whether a type-2 request reflects the user's intention.

1. *Address-bar-entering.* When the user types in a URL in the address bar and hits enter, the request sent by the browser is considered as intended, because we can assure that the user intends to visit the URL she typed in.

   Note that we distinguish between typing in by keyboard and pasting from the clipboard. The adversary may send the victim an email, which contains a URL that links to a CSRF attack. Instead of providing a hyper-link for the user to click, the email can ask the user to copy and paste the URL to the browser's address-bar. To defeat this trick, only when the URL is typed in to the address-bar by the keyboard, the request is intended. If the URL is pasted from the clipboard, the request is not considered to be intended.

2. *Bookmark-clicking.* When the user selects a link from the bookmarks, the request is considered as intended, because users are usually careful in maintaining the bookmarks.

3. *Default-homepage.* When the browser displays the default home page either when it starts or when user clicks the "homepage" button, the request is considered intended, because the configuration of default homepage is set by the user and cannot be easily modified by malicious web sites.

All other type-2 requests are not considered to be intended. For example, when the user clicks a link from the history, or when the user clicks a link outside the browser

(e.g., in an email or a word document), the requests are not considered as intended. When performing those actions, users normally do not have a clear idea about which web site they are going to. The history and the links outside the browser may contain malicious contents that could launch CSRF attacks. Note that these requests are still allowed to proceed, we will only strip sensitive authentication tokens from them.

Browser-enforced *Source-set* Checking. To determine whether a type-1 request reflects the user's intention, we borrow the idea from the server-side referer-checking technique. Our approach has two significant differences. First, the enforcement is done by the browser rather than the web application. In this way, the Referer header does not need to be sent to the web server. This addresses the privacy concerns caused by sending out the Referer header, and it is compatible to the browsers and network devices that block the Referer header. In addition, the browser is able to check the Referer for all requests whose links are provided by a webpage (type-2 requests); so it avoids the dilemma in the server-side referer-checking with the requests that lack a Referer header. Second, we extend the notion of Referer to *Source-set* by taking into account the visual relationships among webpages in the browser. As a result, we can defeat the CSRF attack against Facebook mentioned in Section 6.2.3. *Source-set* checking can only be done in the browser.

Intuitively, the *Source-set* of a request includes all web pages that can potentially affect the request. We define the *Source-set* as follows.

Definition 6.3.1 *The* referer *of a request is the webpage that provides the link to the request. The* Source-set *of a request includes its referer and all webpages hosted in ancestor frames of the referer.*

For example, in Fig. 6.1, when the user clicks the "Post" button in the last tab, a request is sent to Facebook.com. The referer of the request is the innermost iframe that links to `http://www.facebook.com/sharer.php`. The *Source-set* includes the referer and its two ancestor webpages that are from the malicious website (In the at-

tack, the malicious webpage includes an iframe linking to another malicious webpage, which further includes an iframe linking to Facebook.).

The rationale for including all ancestors of the referer page in the *Source-set* of a request is because all ancestor webpages can potentially affect the request. Users are typically unaware of the existence of the frame hierarchy, and they assume they are visiting the website hosted in the top-level frame with the URL shown in the address-bar. The parent frame is able to manipulate the URL, size, position and scrolling of child frame, to fool the user. As a result, when the user performs some actions in the child frame, those actions may not reflect the user's intention. Therefore, the referer and all its ancestor webpages are considered to be in the *Source-set* of a request.

Given a type-1 request, we consider it reflect the user's intention if all webpages in the *Source-set* are from the same website as the destination of the request. This is based on the following assumption: a request sent by a website to itself reflects the user's intention. In other words, a website won't launch a CSRF attack against itself.

6.3.2 Inferring the Sensitive Authentication Tokens

We have introduced a mechanism to infer whether an HTTP request reflects the user's intention. A simple way to defend against the CSRF attacks is to stripe all cookies and other authentication tokens from all requests that may not reflect the user's intention. However, such a policy would break some existing web applications. In particular, it would disable the legitimate cross-site requests that need to carry authentication tokens. An important observation is that although legitimate cross-site requests may need to carry an authentication token, legitimate cross-site requests typically do not lead to sensitive consequence, because sensitive operations typically require an explicit confirmation that is done in the target website. Based on this observation, we further infer whether an authentication token is sensitive or not for a request, and stripe only sensitive authentication tokens from requests that may not reflect the user's intension.

Table 6.2
The default policy of BEAP enforced by the browser

	GET		POST
	HTTP	HTTPS	Sensitive
Session Cookies	Not Sensitive	Sensitive	
Persistent Cookies	Not Sensitive		
HTTP Authorization Header	Sensitive		

We use heuristics derived from analyzing the real-world web applications to determine whether an authentication token is sensitive or not for a request, based on the following information: (1) Whether the request is GET or POST. (2) Whether the token is a session cookie, a persistent cookie or an HTTP authorization header. (3) Whether the communication channel is HTTP or HTTPS. Our heuristics are summarized in Table 6.2 and are explained below.

The HTTP authorization headers are always sensitive. The HTTP authorization headers are typically used in the home/enterprise network. The services using the authorization headers for authentication are typically sensitive, e.g., home router administration, enterprise network services. In addition, it would be severe if a malicious website in the Internet is able to launch a CSRF attack against a service inside the Intranet.

For cookies we distinguish between the two request methods. All cookies that are attached to the POST requests are sensitive for two reasons. First, according to the HTTP/1.1 RFC document, all the operations that have lasting observable effects should be requested using the method POST. Second, the POST requests are used to submit forms and forms are mostly submitted to the same website as that provides the form. So to stripe authentication tokens from the cross-site POST requests will protect all web applications that follow the RFC standard, and won't affect the existing web applications.

However, there exist some web applications that do not follow the standard and use GET requests for sensitive operations. We would like to protect those web applications against the CSRF attacks as well. For the cookies with GET requests, the policy further distinguishes between the session cookies and persistent cookies. The persistent cookies (those that have an expiration date) with GET requests are not sensitive. The persistent cookies are commonly used by the websites to provide personalized services without asking the user to explicitly log in. For example, `Amazon.com` displays recommendations based on the user's history activities. This is achieved by storing the user's identity and related information in persistent cookies. If the user links to `Amazon.com` from a third party website (e.g., a search engine), the request should carry the persistent cookies so that `Amazon.com` is able to recognize the user and provides a personalized service. Therefore, there exists legitimate cross-site GET requests that need to carry persistent cookies. On the other hand, most sensitive web applications (especially financial websites such as banks) use session cookies (those that does not have an expiration date and will be deleted when the browser is closed) as the authentication token for sensitive operations. For example, the persistent cookies are not enough for a user to place an order in `Amazon.com`, he needs to type in his password to obtain a session cookie to place an order. Some financial websites provide a "Remember me" option with the login form, but typically that is used to remember the user's username, the user still needs to type in the password to obtain a session cookie in order to access his account. Furthermore, using persistent cookies for sensitive operations is a bad practice, because the users may access their accounts from public computers (e.g., in an Internet Cafe). Using persistent cookies for authenticating sensitive operations would allow the persons who use the same computer following the user to impersonate the user.

It is a bit complicated for the session cookies with GET requests. We observe some websites issue legitimate cross-site GET requests that need to carry session cookies. In particular, the content sharing websites, such as Digg, Facebook, etc., allow people to discover and share contents from anywhere on the Internet, by submitting links

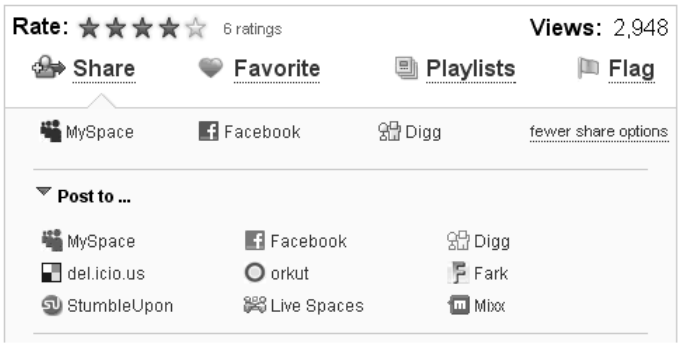

Figure 6.2. Youtube provides links to various content sharing websites under the video.

and stories. Many webpages include links to the submission pages of those websites, so that the users can easily post the current article or video to their accounts. For example, as shown in Figure 6.2, `Youtube.com` provides links to various content sharing websites under each video. When clicking the Facebook link, a GET request is sent from Youtube.com to Facebook.com. If the user already logs in to Facebook.com, the request will carry the session cookie and the user can be directly linked to the submission page (Fig. 6.1(A)) without logging in again. To preserve this functionality of the content sharing websites, the policy treats the session cookies with GET requests using the HTTP protocol as not sensitive. In contract, the session cookies with GET requests using the HTTPS protocol are sensitive, because the sensitive services are typically served over HTTPS.

In conclusion, we infer whether an authentication token is sensitive as summarized in Table 6.2. To defend against the CSRF attack, we stripe the sensitive authentication tokens from the requests that may not reflect the user's intention.

6.4 Security Analysis and Discussions

Effectiveness of BEAP. How effective is BEAP for defending against CSRF attacks? In other words, how effective dose BEAP achieves "all sensitive requests sent

by the browser reflect the user's intention"? We now answer these questions by analyzing under what assumptions the two inferences work correctly.

We observe that, under three assumptions, a CSRF attack always results in a request that BEAP considers to not reflect the user's intention. First, the browser has not been compromised. BEAP is not designed to defend against attacks that exploit vulnerabilities in browsers to take over the browser or the operating system. BEAP defends against CSRF attacks, which exploit web browsers' design feature of allowing cross-site requests. Defending against browser exploitation is orthogonal to our work. Second, a user will not type in a CSRF attack URL in the address bar, or include a CSRF attack page in the bookmark, or use it as the default homepage. Under these two assumptions, type-2 requests that are considered as intended are not CSRF attacks. Third, a website does not include CSRF attacks against itself. This ensures that any CSRF attack via type-1 requests will be correctly classified. The third assumption means that we cannot defend against CSRF attacks that are injected into the target website. For example, the attacker may be able to inject a CSRF attack into a forum via a posting, which sends a posting on the victim's behalf. In this case, the malicious request is actually not a cross-site request, and will be treated as intended. Such an attack cannot be defeated by a pure client-side defense, because the browser cannot tell which requests in a webpage are legitimately added by the web site and which ones are maliciously added by user postings. The problem should be addressed by having the web application sanitize the user input to be displayed in the website, similar to defending against XSS attacks.

Second, BEAP allows non-sensitive cookies to be sent with requests that are not intended. This causes no harm when these requests do not have sensitive consequences. This is true assuming that websites do not contain sensitive operations that (1) use GET requests and rely on persistent cookies for authentication, or (2) use GET requests over HTTP and rely on session cookies for authentication. We would like to point out that these are all bad practices and are vulnerable to attacks other than CSRF attacks. First, using GET for requests that have sensitive consequence

violates the HTTP/1.1 standard [77]. Second, when using persistent cookies for authenticating sensitive services, the accounts can be easily stolen if the user access the account in a public computer. Third, serving sensitive service over HTTP enables the network attacker to launch session injection attack. In particular, we did not observe any financial websites violate these assumptions; they are all hosted over HTTPS and relying on session cookies for authentication.

Compatibility of BEAP. BEAP will stripe cookies and HTTP authentication headers from some requests. Would this affect the existing web applications and change the user's browsing experiences? We now show that the answer is no.

First, we point out that cookie blocking has already been used for other purposes. Cookies, such as those set by `doubleclick.com`, can be used to track users' browsing behavior and violate user' privacy. Because of this, Internet Explorer 6 and later versions protect the user's privacy with respect to cookies [83]. We compared the cookie filtering in IE with BEAP in Section 6.4.1.

It is difficult to use a crawler or an automatic tool to perform a large-scale compatibility testing, because testing the compatibility is possible only when we have an account on a website and log into the account to perform authenticated operations. In particular, creating web accounts on financial websites typically require having physical accounts.

Finally, we note that the functionalities provided by these web sites are not disabled. When cookies are striped, the worst case is that the user needs to re-enter the password in order to perform certain operations.

6.4.1 Compared with IE's Cookie Filtering for Privacy Protection

Cookies are widely used to track users across multiple websites. For example, advertisers typically use cookies to identify the user, so that they can track the user's movement as he visits different publishers. As a result, the advertisers can profile the user's browsing habit and display targeted advertisements.

Table 6.3
The default policy for cookie filtering for privacy protection in IE6

	First-party cookies	Third-party cookies
Persistent cookies with no policy	Leash	Deny
Persistent cookies with unsatisfactory policy	Downgrade	Deny
Persistent cookies with acceptable policy	Accept	Accept
Session cookies	Accept	Treated as persistent cookies

Internet Explorer 6 and later versions protect the user's privacy with respect to cookies [83]. In particular, IE requires web services to deploy policies as defined by P3P (Platform for Privacy Preferences) [84]. When a website does not provide a P3P policy or the policy does not satisfy the user's preference, IE performs cookie filtering against the website. The approach applied by IE's cookie filtering has similarities with our defense against the CSRF attacks, but it aims at protecting privacy while we aim at protecting authenticity.

The cookie filtering infers whether a cookie may violate the user privacy based on the type of the cookie and the heuristics derived from real-world web applications. First, the policy distinguishes between first-party cookies and third-party cookies. A webpage may contain images or other components stored on a third-party website. Cookies that are set during retrieval of these components are called third-party cookies. Third-party cookies are more likely to violate the user privacy because the tracking cookies are typically set when retrieving third-party advertisements. Second, it also distinguishes between session cookies and persistent cookies. For the privacy protection, the persistent cookies are considered to be more dangerous than the session cookies, because the tracking cookies need to persist across multiple sessions.

The default policy (medium) applied by the cookie filtering is shown in Table 6.3 [83]. Depending on the context, IE will accept, deny, downgrade, or leash the cookie. A downgraded cookie is a persistent cookie that is removed when the session ends or the cookie expires, whichever comes first. A leashed cookie is one that is sent only on requests to download first-party content. When requests are made for

third-party content, these cookies are suppressed. The leashed cookie is similar to the sensitive authentication tokens that are suppressed from the requests that may not reflect the user's intention in our proposal.

7 SUMMARY

In access control systems, a request is issued by the subject, while the privileges are granted to principals. An essential step in performing the access control check is to link the subjects in the requests to the principals in the policies. We introduce the notion of origins of a request. The origins of a request is the set of principals that cause the subject to issue the request and thus should be responsible for the request.

Many access control systems are vulnerable to certain attacks because their limitations in identifying the origins. We targeted two real-world access control systems, operating system access control and browser access control. The discretionary access control used in today's operating systems is vulnerable to the vulnerability exploitation and Trojan horse attacks. The same origin policy (SoP) model implemented in modern browsers are vulnerable to the pretty-bad-proxy adversary for HTTPS and the cross-site request forgery attack. We have showed that these access control mechanisms are vulnerable because they did not properly identify the origins of the requests.

To enhance the DAC to defend against remote exploitation, we introduced the UMIP model, a simple and practical MAC model for host integrity protection that defends against attacks targeting network server and client programs. The key idea of UMIP is to associate each process with an integrity level, which indicates whether the process may have been exploited by the remote attackers. By adding this bit to the origins of the request, UMIP is able to defend against the remote exploitation attacks. UMIP supports existing applications and system administration practices, and has a simple policy configuration interface.

To enhance the DAC to defend against Trojan horses, we proposed the IFEDAC model, which uses information flow techniques to track which principals are the origins of a request, thereby achieving the DAC policy without assuming that software

are bug-free and benign. While using techniques from mandatory information flow, IFEDAC follows the discretionary control principle and allows owners to decide which other users can access the file and uses the identities of the requester to decide access. In this sense, IFEDAC is the first DAC model that can defend against Trojan horses.

With a focused examination of the PBP adversary against the HTTPS deployment in modern browsers, we discovered a set of PBP-exploitable vulnerabilities in IE, Firefox, Opera, Chrome browsers and many websites. The existence of the vulnerabilities clearly undermines the end-to-end security guarantees of HTTPS. The PBP adversary is able to inject a malicious web message into a HTTPS session and the browser will accept the malicious message as it comes from the target HTTPS domain. These vulnerabilities should be fixed by correctly identifying the origin of every web message in the HTTPS sessions and only accept those coming from the target HTTPS domain and transmitted over a secure connection. We have reported the vulnerabilities to major browser venders. All vulnerabilities are acknowledged and the fixes have been implemented or proposed.

To defeat the cross-site request forgery attacks, we have proposed a browser-based mechanism called BEAP. The CSRF attack is feasible because the same origin policy model did not restrict the operation of sending authentication tokens and the browser assumes every request reflects the user's intention. The reality is that the browser can be fooled to send malicious requests by the malicious webpages. BEAP defeats the CSRF attacks by identifying the true origins of a request sent by the browser and inferring whether the request reflects the user's intention. BEAP strips the sensitive authentication tokens from the requests that may not reflect the user's intention. BEAP can effectively defend against the CSRF attacks, and does not break the existing web applications.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Ross Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[2] DOD. *Trusted Computer System Evaluation Criteria*. Department of Defense 5200.28-STD, December 1985.

[3] NCSC. National Computer Security Center: A guide to understanding discretionary access control in trusted systems, September 1987. NCSC-TG-003.

[4] SUN. Same origin policy for JavaScript. `https://developer.mozilla.org/En/Same_origin_policy_for_JavaScript`.

[5] Peter Watkins. Cross-site request forgery, 2001. `http://www.tux.org/~peterw/csrf.txt`.

[6] Chris Shiflett. Foiling cross-site attacks, October 2001. `http://shiflett.org/articles/foiling-cross-site-attacks`.

[7] Chris Shiflett. Security corner: Cross-site request forgeries, December 2004. `http://shiflett.org/articles/cross-site-request-forgeries`.

[8] Security-enhanced Linux. `http://www.nsa.gov/selinux`.

[9] Apparmor application security for linux. `http://www.novell.com/linux/security/apparmor/`.

[10] The advantages of running applications on Windows Vista. `http://msdn2.microsoft.com/en-us/library/bb188739.aspx`.

[11] NSA. Security-enhanced Linux. `http://www.nsa.gov/selinux/`.

[12] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil D. Gligor. Subdomain: Parsimonious server security. In *Proceedings of the 14th Conference on Systems Administration (LISA 2000)*, pages 355–368, December 2000.

[13] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[14] Deborah D. Downs, Jerzy R. Rub, Kenneth C. Kung, and Carole S. Jordan. Issues in discretionary access control. In *Proceedings of IEEE Symposium on Research in Security and Privacy*, pages 208–218, April 1985.

[15] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, Mitre Corporation, March 1976.

[16] M. D. Mcllroy and J. A. Reeds. Multilevel security in the unix tradition. *Software—Practice and Experience*, 22(8):673–694, August 1992.

[17] N. Provos. Improving host security with system call policies. In *Proceedings of the 2003 USENIX Security Symposium*, pages 252–272, August 2003.

[18] Freebsd.org. Frequently asked questions for freebsd 4.x, 5.x, and 6.x. `http://www.freebsd.org/doc/en_US.ISO8859-1/books/faq/`.

[19] LIDS: Linux intrusion detection system. `http://www.lids.org/`.

[20] David R. Wichers, Douglas M. Cook, Ronald A. Olsson, John Crossley, Paul Kerchen, Karl N. Levitt, and Raymong Lo. Pacl's: An access control list approach to anti-viral security. In *Proceedings of the 13th National Computer Security Conference*, pages 340–349, October 1990.

[21] K. J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE, April 1977.

[22] T. Fraser. LOMAC: Low water-mark integrity protection for COTS environments. In *2000 IEEE Symposium on Security and Privacy*, May 2000.

[23] David D. Clark and David R. Wilson. A comparision of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, May 1987.

[24] Paul A. Karger. Implementing commercial data integrity with secure capabilities. In *Proceeding IEEE Symposium on Security and Privacy*, pages 130–139, 1988.

[25] Theodore M. P. Lee. Using mandatory integrity to enforce "commercial" security. In *Proceeding IEEE Symposium on Security and Privacy*, pages 140–146, 1988.

[26] Catherine Jensen McCollum, Judith R. Messing, and LouAnna Notargiacomo. Beyond the pale of MAC and DAC–defining new forms of access control. In *Proceeding IEEE Symposium on Security and Privacy*, pages 190–200. IEEE Computer Society, 1990.

[27] Boniface Hicks, Sandra Rueda, Trent Jaeger, and Patrick McDaniel. From trusted to secure: Building and executing applications that enforce system security. In *Proceedings of the USENIX Annual Technical Conference*, June 2007.

[28] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2005.

[29] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceedings of the 15th USENIX Security Symposium*, August 2006.

[30] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, October 1993.

[31] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, and R. Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 2005 ACM Symposium on Operating System Principles*, October 2005.

[32] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazires. Making information flow explicit in histar. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[33] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[34] Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceeding USENIX Security Symposium*, August 2005.

[35] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *Proceeding USENIX Security Symposium*, July 2006.

[36] Shuo Chen, David Ross, and Yi-Min Wang. An analysis of browser domain-isolation bugs and a light-weight transparent defense mechanism. In *Proceeding ACM Conference on Computer and Communications Security (CCS)*, November 2007.

[37] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Md5 considered harmful today – creating a rogue ca certificate. `http://www.win.tue.nl/hashclash/rogue-ca/#sec71`.

[38] Stuart E. Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. The emperor's new security indicators: An evaluation of website authentication and the effect of role playing on usability studies. In *Proceeding IEEE Symposium on Security and Privacy*, May 2007.

[39] Shuo Chen, Jose Meseguer, Ralf Sasse, Helen J. Wang, and Yi-Min Wang. A systematic approach to uncover security flaws in gui logic. In *Proceeding IEEE Symposium on Security and Privacy*, May 2007.

[40] Collin Jackson and Adam Barth. Forcehttps: Protecting high-security web sites from network attacks. In *WWW '08: Proceedings of the 16th international conference on World Wide Web*, 2008.

[41] Chris Karlof, Umesh Shankar, J.D. Tygar, and David Wagner. Dynamic pharming attacks and locked same-origin policies for web browsers. In *Proceeding ACM Conference on Computer and Communications Security (CCS)*, November 2007.

[42] US-CERT. Multiple dns implementations vulnerable to cache poisoning. `http://www.kb.cert.org/vuls/id/800113`.

[43] Grant Bugher. Wpad: Internet Explorer's worst feature. `http://perimetergrid.com/wp/2008/01/11/wpad/`.

[44] Niels Teusink. Hacking random clients using wpad. http://blog.teusink.net/2008/11/about-two-weeks-ago-i-registered-wpad.html.

[45] Andreas Pashalidis. A cautionary note on automatic proxy configuration. In *Proceedings of the IASTED International Conference on Communication, Network, and Information Security*, 2003.

[46] M. Johns and J. Winter. RequestRodeo: Client side protetion against session riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.

[47] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceeding ACM Conference on Computer and Communications Security (CCS)*, October 2008.

[48] N. Jovanvoic, E. Kirda, and C. Kruegel. Preventing cross site request forgery attacks. In *Proceedings of the Second IEEE Conference on Security and Privacy in Communication Networks*, September 2006.

[49] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceeding USENIX Security Symposium*, pages 1–13, June 1996.

[50] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. Practical domain and type enforcement for UNIX. In *Proceeding IEEE Symposium on Security and Privacy*, pages 66–77, May 1995.

[51] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A domain and type enforcement UNIX prototype. In *Proceeding USENIX Security Symposium*, June 1995.

[52] Ravi Sandhu. Good-enough security: Toward a pragmatic business-driven discipline. *IEEE Internet Computing*, 7(1):66–68, January 2003.

[53] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley Professional, 2003.

[54] D. Brumley and D. Song. PrivTrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, August 2004.

[55] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 2003 USENIX Security Symposium*, pages 231–242, August 2003.

[56] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX track: USENIX Annual Technical Conference*, pages 29–42, June 2001.

[57] T. Dierks and E. Resorla. Rfc5246: The transport layer security (tls) protocol. http://tools.ietf.org/html/rfc5246.

[58] Jason Rafail. Cross-site scripting vulnerabilities. http://www.cert.org/archive/pdf/cross_site_scripting.pdf.

[59] Cookie property. http://msdn2.microsoft.com/en-us/library/ms533693.aspx.

[60] Internet access configuration instructions for some conferences, hotels, hospitals and airports that require proxies. http://homepage.eircom.net/~acsi/encs08.htm, http://www.hw.ac.uk/uics/Help_FAQs/WiFi_FAQs.html, http://www.vistagate.com/Demo/Administration/Help.htm, http://www.ucd.ie/mcri/resources/IP_logistics_students.pdf, http://www.grh.org/patWireless.html.

[61] Internet access configuration instructions for some university departments and libraries that require proxies. http://groups.haas.berkeley.edu/hcs/howdoi/AirBearsXP.pdf, http://www.lib.berkeley.edu/Help/proxy_setup_ie5-7_dialup.html, http://physics.ucsd.edu/~sps/html/resources/articles/sciamsetup.html, http://www.lib.ucdavis.edu/ul/services/connect/proxy/step1/iewindowslong.php.

[62] Anonymizer: free web proxy, cgi proxy list, free anonymizers and the list of web anonymizers list. http://www.freeproxy.ru/en/free_proxy/cgi-proxy.htm.

[63] MSDN Online. Using automatic configuration, automatic proxy, and automatic detection. http://www.microsoft.com/technet/prodtechnol/ie/reskit/6/part6/c26ie6rk.mspx?mfr=true.

[64] MSDN Online. Winhttp autoproxy support. http://msdn.microsoft.com/en-us/library/aa384240.aspx.

[65] Winpcap: The windows packet capture library. http://www.winpcap.org/.

[66] Andrea Bittau, Mark Handley, and Joshua Lackey. The final nail in wep's coffin. In *Proceeding IEEE Symposium on Security and Privacy*, May 2006.

[67] Network load balancing: Frequently asked questions for windows 2000 and windows server 2003. http://technet2.microsoft.com/windowsserver/en/library/884c727d-6083-4265-ac1d-b5e66b68281a1033.mspx?mfr=true.

[68] The web hacking incidents database, 2008. http://www.webappsec.org/projects/whid/byid_id_2008-05.shtml.

[69] OWASP. Top ten most critical web application security vulnerabilties. Whitepaper, 2007. http://www.owasp.org/index.php/Top_10_2007.

[70] US-CERT. Cross-site request forgery (CSRF) vulnerability in the Linksys wrt54gl wireless-g broadband router. CVE-2008-0228, January 2008. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-0228.

[71] US-CERT. Cross-site request forgery (CSRF) vulnerability in @mail webmail 4.51. CVE-2006-6701, December 2006. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-6701.

[72] US-CERT. Multiple cross-site request forgery (CSRF) vulnerabilities in phpmyadmin before 2.9.1. CVE-2006-5116, October 2006. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2006-5116.

[73] US-CERT. Google gmail cross-site request forgery vulnerability. Vulnerability Note 571584, October 2007. http://www.kb.cert.org/vuls/id/571584.

[74] Kelly Jackson Higgins. CSRF vulnerability: A 'sleeping giant', 2006. http://www.darkreading.com/document.asp?doc_id=107651.

[75] D. Kristol and L. Montulli. HTTP state management mechanism. RFC 2965, October 2000. http://www.ietf.org/rfc/rfc2965.txt.

[76] J. Franks, P. Hallam-Baker, J.Hostetler, S.Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP authentication: Basic and digest access authentication. RFC 2617, June 1999. http://www.ietf.org/rfc/rfc2617.txt.

[77] Network Working Group. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999. http://www.ietf.org/rfc/rfc2616.txt.

[78] phpBB. Create communities worldwide. http://www.phpbb.com.

[79] US-CERT. Cross-site request forgery (CSRF) vulnerability in privmsg.php in phpbb 2.0.22. CVE-2008-0471, January 2008. http://nvd.nist.gov/nvd.cfm?cvename=CVE-2008-0471.

[80] Robert Hansen and Tom Stracener. Xploiting google gadgets: Gmalware and beyond, August 2008.

[81] P. Koch. Frame busting. http://www.quirksmode.org/js/framebust.html.

[82] Collin Jackson. Defeating frame busting techniques, 2005. http://www.crypto.stanford.edu/framebust/.

[83] MSDN. Privacy in internet explorer 6. http://msdn.microsoft.com/en-us/library/ms537343({VS}.85).aspx.

[84] The platform for privacy preferences project (p3p). http://www.w3.org/TR/P3P.

VITA

VITA

Ziqing Mao received his B.E. degree in computer science from Tsinghua University, People's Republic of China in 2005. Since then he has been working toward a Ph.D. degree in Department of Computer Science at Purdue University. He received the M.S. degree in computer science in 2003. His research interest lies in computer security in general, with a focus on access control, operating system security, web security and browser security.