Information-Flow Control for Web Application Security

情報フロー制御による
安全なWebアプリケーション

by
Sachiko Yoshihama

June 2006

A doctorial dissertation submitted to

the Graduate School of Environment and Information Sciences,

Yokohama National University

in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

New trends in Web technologies and use-cases are invalidating the traditional security models such as the browser security model and the perimeter defense approach, and call for changes in the security models of the Web. This thesis proposes security mechanisms for the Web environment based on information flow control. Information-Flow Control (IFC) is a fundamental requirement for information security. The goal of IFC is to protect information to satisfy two fundamental properties, *confidentiality* and *integrity*. Confidentiality means protecting sensitive data from being disclosed to unauthorized entities. Integrity means protecting data from unauthorized modification. IFC is also a very essential requirement of the Web. Since many existing applications have migrated to Web-based interfaces, confidential and potentially high-value data are under risk of exposure or compromise because of human errors or vulnerabilities in Web applications and systems. In addition, the boundary between the data and executable programs is becoming obscured in the newer Web technologies, and data is often interpreted as part of a program. Because the integrity of data also governs the integrity of the Web application behavior, it is important to control the application by taking the integrity of the data into account. History has shown that application-level countermeasures are error prone, and thus it is important that the runtime

environment provides security protection.

This thesis first describes the various threats to Web applications, and then makes three technical proposals to mitigate the risks from these threats.

The first proposal is a new secure browser model that mitigates the design flaws of the same-origin policy and thus mitigates the threats to Web applications. In the proposed browser model, all data, i.e., content received from servers, is associated with a security label that identifies the origin of the data. The label represents the security domain of the URL from which the content originated. In addition, the user and each content provider can define the access control policy to control how content (i.e., script) can access resources, such as the browser's internal DOM tree or the document cookies. The access control policy can controls execution of script as well as network access by JavaScript and HTML. The proposed access control model is built on top of information-flow-based access control, which judges the privilege of the content based on the origin of the data used in each operation. The origin of the data is tracked through script execution. The proposed model can enforce access control policies based on the originator of an action even in dynamic and self-mutating client-side Web applications.

The second proposal is a technology called the Dynamic Information Flow Control Architecture (DIFCA). DIFCA achieves information flow control by modifying the application code with Java byte-code instrumentation to insert inline monitoring code into the application code. DIFCA can track fine-grained information-flows that are caused by every instruction in a running application, taking the dynamic runtime conditions into account. It does not require any source code of the target application, and is independent

of the Java VM implementation. In addition, it effectively labels data for input from or output to external environments, especially data exchanged with databases, and supports fine-grained application-level policies, including declassification policies.

The third proposal is a data leakage prevention system by using Web proxies, called Web DLP. The technology consists of the protocol and the content analysis to identify sensitive data flows across domains (such as data flows from the on-premise domain to an off-premise domain, or between multiple off-premise domains), detection of the sensitivity of the data, and prevention of undesirable data flows. The proposed technology addresses the data flow risks when the modification of the client-side or the server-side runtime environment is not possible or realistic.

Acknowledgments

I am deeply grateful to Professor Tsutomu Matsumoto of Yokohama National University for his support and guidence through my Ph.D. studies. I wish to extend my deep appreciation to my dissertation committee, Professor Naoyoshi Tamura, Professor Tomoharu Nagao, Professor Tatsunori Mori, and Associate Professor Junji Shikata. I would also like to express appreciation to Assistant Professor Katsunari Yoshioka and my fellow students in the Matsumoto Laboratory for their support and fruitful discussions.

Colleagues at IBM Research - Tokyo have always been stimulating and supportive. I am particularly grateful to Yuji Watanabe, Takuya Mishina, Ai Ishida, Takaaki Tateishi, Naoshi Tabuchi, Takeo Yoshizawa, Satoshi Makino, Shinya Kawanaka, Masayoshi Teraguchi, Takahide Nogayama, and Frederik De Keukelaere. Some of the work in this dissertation was done jointly with them, and I owe them for their inspiration and support. I also very much appreciated the support from my current and past managers, especially Hiroshi Maruyama, Michiharu Kudo, Naohiko Uramoto, Kazuya Kosaka, and Tamiya Onodera. I would like to express my appreciation to Shannon Jacobs in IBM Japan for his patience and insightful suggestions for this dissertation and some of my earlier papers. I would also like to thank my colleagues at IBM T.J. Watson Research Center, in particular Paul B. Chou, Jennifer Lai,

Thomas Bridgman, Anthony Levas, and Danny Wong, for their support and encouragement. Marisa Viveros and Mayumi Itakura have been guiding me as mentors, and I appreciated their encouragement in pursuing my career at IBM Research.

Part of work in this dissertation was done while I was studying in the Institute of Information Security in Yokohama, Japan. I am sincerely grateful for guidance and support from Professor Kazuko Oyanagi. Part of the work in this dissertation was sponsored by the Ministry of Economy, Trade and Industry, Japan (METI) under a contract for the New-Generation Information Security R&D Program.

Finally, I would like to thank my mother and my sister, Masako and Kumiko Yoshihama, and my partner Naoki Sugimoto for their continuous love and support.

Contents

Chapter 1

Introduction

1.1 Background

The World Wide Web has been changing its role since its emergence around 1990. In its early days, the Web was a technology to browse mostly static content on the servers using client-site software called "Web browsers", and the users of the technology were limited to those in a relatively closed research community.

The technology is still evolving to add dynamic behaviors and flexibility. Web servers have become *Web application servers*, which receive HTTP requests and return dynamically generated content, changing the Web from a content browsing system to a client-server application system.

In 1996, Netscape 2.0, the most widely used Web browser around that time, introduced JavaScript to enable scripting capabilities in the Web browser. Although JavaScript was used only supplementarily in the beginning, it gradually increased its importance over time. A new Web application model that was enabled by the rich use of JavaScript was initially called Dynamic HTML

(DHTML), and then Ajax (Asynchronous JavaScript and XML), drastically improved the user experience in Web applications.

At the same time, the use cases of the Web have been shifting from academic uses to more business-oriented applications. Many business entities today provide their services via the Web interfaces, and many existing legacy applications were *Web enabled* by adding Web-based user interfaces. Such changes made Web applications attractive targets for attackers.

The buzzword *Web 2.0* symbolizes a trend of the Web that is epitomized by user-generated content and service *Mashups*, supported by Ajax. Software as a Service (SaaS) is another trend that is drawing strong attention recently. In SaaS, application software is hosted on a service provider's infrastructure and provided to the customers as a service, and the cost is charged as utility bills. Because it can reduce the initial investment and the maintenance costs, SaaS is gaining widespread acceptance.

These trends call for changes in the security of the Web. Traditionally, the trust boundaries of Web applications were identified by the DNS domain name of the Web servers, and the Web browsers enforced the simple isolation policy called the *Same-Origin Policy* for those boundaries. However, user-generated content and mashups invalidated this simple security model, because now a Web application that is hosted on a trusted Web server may include content from any other resource including untrustworthy ones.

These trends also invalidated the traditional perimeter defense by using firewalls. The major security risks of the Web used to be represented by the attacks against the Web servers, but now a Web browser can be used as a stepping stone to attack intranet resources, by using injected and malicious

client-side JavaScript. SaaS and hosted applications increased the risks of the leakage of sensitive data from a Web browser to external servers, either because of users' mishandling or because of attacks by malicious adversaries.

1.2 Information-Flow Control for Web Application Security

This thesis proposes a series of security mechanisms for the Web environment based on information flow control.

Information-Flow Control (IFC) is a fundamental requirement for information security. The goal of IFC is to protect information and to satisfy two fundamental properties, *confidentiality* and *integrity*. Confidentiality means protecting sensitive data from being disclosed to unauthorized entities. Integrity means that data cannot be modified without authorization, and unauthorized modification can be detected.

IFC is also a very essential requirement for the Web. Since many existing applications are migrating to Web-based interfaces, confidential and potentially high-value data is under the risk of exposure and compromise, due to vulnerabilities and human errors. In addition, the boundary between the data and executable programs is obscure in the Web technologies, and data is often interpreted as part of a program. Because the integrity of the data also affects the integrity of the Web application behavior, it is important to control the application by taking the integrity of the data into account.

History has shown that the application-level countermeasures are prone to errors, and thus it is important that the runtime environment provides

security protection. This thesis proposes three technologies to mitigate risks from various threats in the Web. The proposal consists of technologies to remodel the client and the server-side runtime environment, to track information flows in the application code to protect the confidentiality and integrity of data. as well as to control the application behavior (theme 1 and theme 2). In addition, the proposals include a technology to detect and prevent information flow over the HTTP protocol, and to mitigate the risks when it is not possible or realistic to change the runtime environments (theme 3).

1.3 Organization of this Thesis

The rest of this thesis is organized as follows.

Chapter 2 describes the overview of the various threats for the Web applications, and discusses how the proposed technologies address these threats [96].

Chapter 3 reviews the existing technologies and clarifies the contribution of this thesis.

Chapter 4 describes Theme 1 [100] [99], a proposal for a new secure browser model that addresses the design flaws of the same-origin policy and mitigates the threats to Web applications. In the proposed browser model, all data, i.e., content received from servers, is associated with the *security label* that identifies the origin of the data. The label represents the security domain of the URL from which the content originated. In addition, the user and each content provider can define the *access control policy* to control how content (i.e., scripts) can access resources, such as the browser's internal DOM tree and the document cookies. The access control policy can also

control the execution of script as well as network access by JavaScript and HTML. The proposed access control model is built on top of information-flow-based access control, which judges the privilege of the content based on the origin of the data used in each operation. The origin of the data is tracked through script execution. The proposed model can enforce access control policies based on the originator of an action even in dynamic and self-mutating client-side Web applications.

Chapter 5 describes Theme 2 [98] [101] [97], which is the Dynamic Information Flow Control Architecture (DIFCA) for Web servers. DIFCA achieves information flow control by modifying the application code by *Java byte-code instrumentation* to insert inline monitoring code into the application code. DIFCA can track fine-grained information-flows that are caused by each instruction in a running application, taking the dynamic runtime conditions into account. It does not require any source code of the target application, and being independent of Java VM implementations. In addition, it effectively labels data for input from or output to external environments, especially data exchanged with databases, and supports fine-grained application-level policies, including declassification policies.

Chapter 6 describes Theme 3, which is the Web-based Data Leakage Prevention. The technology consists of the protocol and the content analysis to identify sensitive data flows across domains (such as data flows from the on-premise domain to an off-premise domain, or between multiple off-premise domains), to assess the sensitivity of the data, and then to prevent undesirable data flows. The proposed technology addresses the data flow risks when the modification of the client-side or the server-side runtime environment is

not possible or realistic.

Chapter 7 concludes the thesis.

Chapter 2

Analysis of Web Application Threats

To understand the full picture of Web application security threats, this chapter introduces basic concepts of Web application security, defines models and classes of threats, describes major attack vectors, and then gives an overview of contribution of this thesis.

The rest of this chapter is organized as follows.

Section 2.1 reviews the basics of the security of the Web, such as the Same-Origin Policy, technical characteristics of Web 2.0, and compare the characteristics of threats in Web 2.0 to those in traditional Web 1.0.

Section 2.2 defines models of Web application threats.

Section 2.3 defines categories of the Web application threats.

Section 2.5 discusses the foci of this work.

Section 2.4 describes the details of the threats that are focus of this thesis work.

2.1 Web Security Basics

This section introduces the basics of the security of the Web. In particular, this section will describe the Same-Origin Policy, a de facto standard security model that is used by most Web browsers. Then this section will discusses the new technology trend of the Web called *Web 2.0* and its technical characteristics. Finally, the characteristics of threats in Web 2.0 are compared to those in traditional Web 1.0 applications.

2.1.1 The Same-Origin Policy

When content from multiple originators is integrated into a single application, some of the content might have a different level of trust from other or some content may be untrustworthy A natural requirement is to isolate content from each originator to minimize the interference.

The same-origin policy is the part of the current browser's protection mechanism that isolates Web applications coming from different domains under the assumption that domains represent originators. That is, if applications in multiple windows or frames are downloaded from different servers, they should not be able to access each other's data and scripts. Note that the same-origin policy only applies to HTML documents. JavaScript files that are imported into an HTML document with `<script src="..." >` tags are regarded as part of the same-origin as the HTML document. This policy is implemented in all of the major browser implementations.

In the context of XMLHttpRequest, the same-origin policy is intended to control an application's interaction with remote servers. However, the same-origin policy has only limited impact on Web 2.0 applications for several

reasons:

- Browsers enforcing the same-origin policy check the domain name of the server as a string literal. For example, `http://www.abc.com/` and `http://12.34.56.78/` are treated as different domains even if the IP address of `www.abc.com` is actually `12.34.56.78`. In addition, any path expression of the URL is ignored. For example, `http://www.abc.com/~alice` is identified as the same-origin as `http://www.abc.com/~malroy`, ignoring the fact that different directories may belong to different users.

- Most Web browsers allow a Web application to relax its domain definition to the super-domain of the application itself. For example, if the application is downloaded from `www.abc.com`, the application can override the document.domain property to `abc.com` or just `com` (for example, some older versions of Firefox). Most of the latest browsers only allow access to window objects between windows or frames that have overridden their document.domain property to the same value. However, some older versions of browsers still in use allow making XMLHttpRequest connections to the domain specified by the `document.domain` property.

- Even if a Web server is in the trusted domain, it might not be the originator of the content, especially in the context of Web 2.0. For example, an enterprise portal server, Web-based mail server, or wiki may be trusted, but the content being hosted may include input from potentially malicious third parties, which could be the target of cross-site scripting (XSS) attacks (described in Section 2.4.3). Therefore, the

domains of the servers do not always represent the trustworthiness of the content.

2.1.2 Technical Characteristics of Web 2.0

This section introduces two technical characteristics of Web 2.0: *Ajax* and *Mashups*.

Ajax

Although there are no clear boundaries between Web 2.0 and the traditional Web (often called Web 1.0), it seems safe to say that the most significant technical difference between them is a programming model called *Ajax*. Ajax stands for *Asynchronous JavaScript* and *XML*, and is characterized by dynamic page updates and asynchronous communications using JavaScript, the most popular scripting language for Web-client-side use.

Figure 2.1 shows a comparison between the Web 1.0 and Ajax interaction models. In traditional Web 1.0 applications, an HTML document downloaded from a URL is displayed on the Web browser basically as it is. If the HTML document includes an HTML form, the user would input data into the form, and the data is sent to the server when the submit button is clicked. The server generates a new HTML document, possibly utilizing some of the user input data, and the document is downloaded by the Web browser to be presented to the user as the next state of the user interface. As illustrated in this example, interactions between the user and the Web browser as well as the communications between the browser and the Web servers are synchronized. Although JavaScript is also used in Web 1.0 applications, its usage

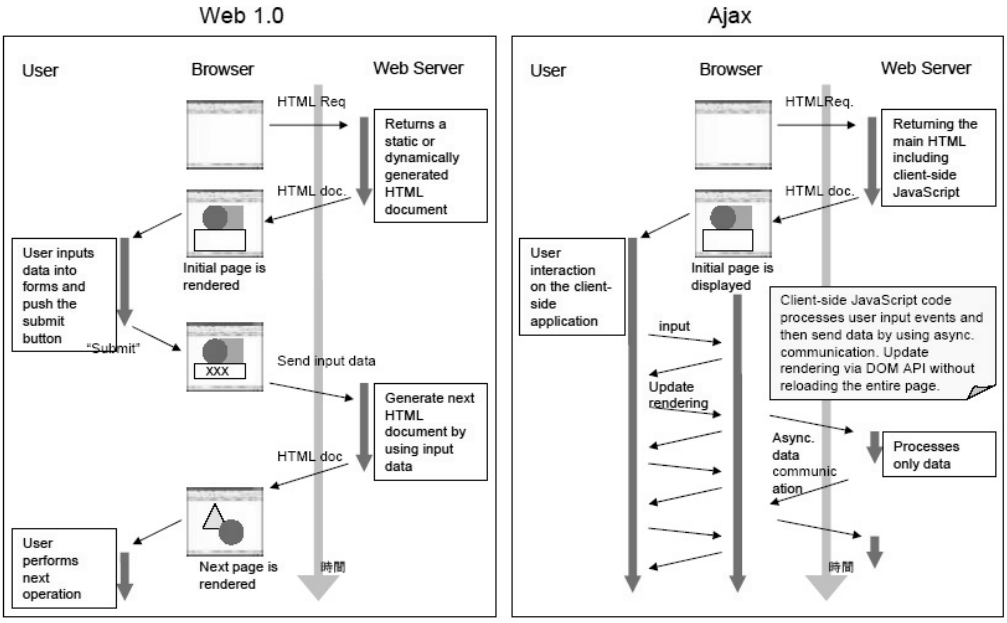


Figure 2.1: Comparison of Interaction Model in Web 1.0 and Ajax

was supplemental, such as minor enhancements for the user interactions.

In contrast, in a typical Ajax application, the initial HTML document consists mainly of a set of client-side program written in JavaScript. After the browser downloads and renders the HTML, the client-side JavaScript code communicates with the server asynchronously, such as by using the XMLHttpRequest browser API, to download XML or JSON data. Most of today's browsers hold the content of the webpage in a tree-structured representation called the Document Object Model (DOM). The client-side JavaScript code can access the DOM tree by using the browser's built-in APIs, to read data from the DOM, or to update the UI by writing data into the DOM. User interaction events such as keyboard or mouse events are handled by the client-side JavaScript code using the event propagation

mechanism of the DOM.

By using these mechanisms, Ajax applications can handle asynchronous interactions between the user and the browser and between the browser and the Web server, resulting in an improved user experience that is as good as native desktop applications.

Mashups

The use of *Mashups* is the second major technical feature of Web 2.0. Mashups integrate existing services or content to create new integrated applications. For example, an on-line restaurant guide can incorporate a map service as well as user reviews and blogs to provide an integrated restaurant guide with location-based services and user-reported reputations.

2.1.3 Technical Characteristics of Threats against Web 2.0 Applications

The technical differences and the trends of attacks in Web 1.0 and Web 2.0 can be described as follows.

In Web 1.0, the server-generated HTML is displayed on the browser almost as sent, and JavaScript is used only for supplements. In Web 2.0, the client-side JavaScript is richer, and plays a major role in the user experience and the UI rendering.

Hyper-Text Transport Protocol (HTTP) is the major protocol for exchanging Web content. In Web 1.0, HTTP is mainly used for transporting HTML documents and images that are visible to the user. In Web 2.0, HTTP is also used to exchange information that may not be shown to the user.

For example, HTTP can be used to submit a command from the client-side JavaScript on a browser to a server, or to control JSON data exchange.

As a result, there are differences in the types of attacks possible. In Web 1.0, most of the sensitive resources are on the Web server, and thus preventing illegal access to the server is the major goal of Web application security. User authentication, authorization, and proper session management are the keys for security.

In Web 2.0, however, the sensitive resources exist not only on the Web server but on the Web browsers. For example, a typical XSS attack in a Web 1.0 application is to steal session information stored in cookies to impersonate the legitimate user (i.e., session hijacking). In contrast, in Web 2.0, an injected script may not only steal session information but also listen to DOM keyboard events to steal user keystrokes from the browser. In the keystroke-monitoring case, since the user key input is the target of the attack, the attack succeeds without accessing any resources on the Web server.

Starting with an example, a typical attack against session management in Web 1.0 is to predict or steal session information to hijack an authorized session. For the corresponding example of a *Cross-Site Request Forgery (CSRF)*, a typical attack to authorization in Web 2.0, there is no need for the attacker to obtain session information. Instead, the attacker may execute malicious script on the victim user's Web browser, and thus issue commands to the Web server by *free-riding* on a legitimate authorized session. Section 2.4.6 will discuss CSRF in more detail.

Local Network Attacks are a new category of attacks to attacks on devices in the same local network as the victim, using the victim's browser as

a stepping stone. Since the target of the Local Network Attack is not necessarily a Web server, the attack cannot be prevented by using the session management mechanisms of the Web server. Section 2.4.7 will discuss Local Network Attacks in more detail.

These examples show how threats against Web 2.0 applications can be characterized as attacks that exploit powerful client-side capabilities.

2.2 Threat Models

This section defines models of Web application threats from two perspectives. Section 2.2.1 defines the entities involved in threat scenarios as well as the types of attack paths. Section 2.2.2 describes the types of effects of the attacks.

2.2.1 Entities and Paths

The first perspective of the threat model can be characterized by the entities that are involved in each attack as well as the types of the attack paths as illustrated in Figure 2.2.

There are five types of entities in the Web application threat scenarios:

- **Honest Users.** Honest users are potential targets for attacks. Although honest users do not have malicious intent, such a user's browser can be used by an attacker as a stepping stone, by running malicious JavaScript code in the browser.

- **Honest Servers.** Honest servers are also potential targets for attacks. An honest server does not perform malicious acts, but may have vul-

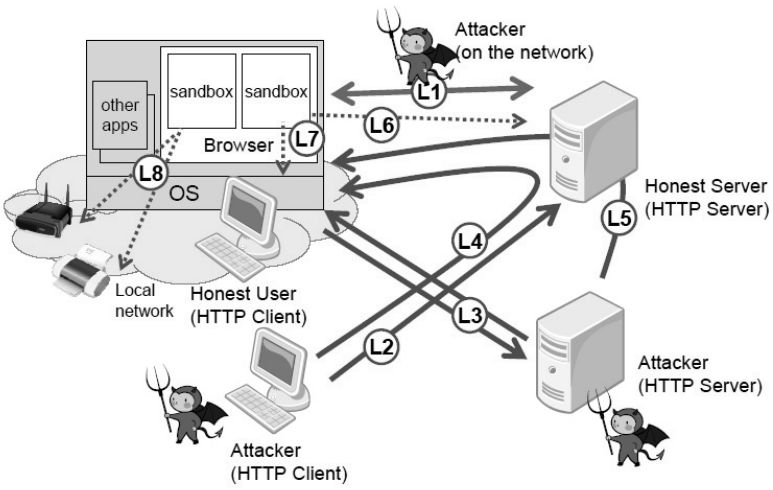


Figure 2.2: Entities and Paths of Threats

nerabilities that can be exploited by attackers. For example, an honest server often host malicious content as a result of cross-site scripting vulnerabilities.

- **Attackers.** Attackers who act as Web users and aim at attacking honest servers, users, or both. For example, an attacker can issue manipulated HTTP requests to Web servers by using various tools, or may eavesdrop on or compromise the communications between honest users and honest servers if the communication channels are not properly protected.

- **Attackers' Servers.** Web servers with malicious intentions, which may host fraudulent services, or provide malicious content to an honest server for mashups.

- **Local Network Devices.** The devices that are at the honest user's premises, such as network printers, routers, or other servers that are

typically in the same local network as the user and protected from external networks by firewalls.

There are eight paths for threats. The paths L1 to L5 are the routes by which malicious data is transferred from an attacker to the target. When the attack occurs via a user's Web browser, the content running in the browser can attack other entities through the paths L6, L7, and L8.

- L1. Threats against communication channels between an honest user and an honest server.
- L2. Threats against honest servers by an attacker.
- L3. Threats against honest users by an attacker's server.
- L4. Threats against honest users by an attacker via an honest server. (Cross-Site Attacks)
- L5. Threats against honest users by an attacker's server via an honest server. (Cross-Site Attacks by mashups)
- L6. Threats against honest servers via an honest user's Web browser.
- L7. Threats against an honest user's computer via the user's Web browser.
- L8. Threats against the local network devices by using an honest user's Web browser as a stepping stone.

2.2.2 The Effects of Threats

The second perspective of the threat model can be characterized by the effects of the threats. The targets of attacks can be categorized into five primary types:

- **Confidentiality:** To steal sensitive information.
- **Integrity:** To modify or alter the data or behavior of Web applications.
- **Identity:** To allow an attacker to impersonate an honest user.
- **Authorization:** To circumvent authorization and perform unauthorized actions.
- **Availability:** Denial of Services (DoS) attacks that prevent honest servers from providing services, or prevent honest users from using their computers or browsers.

Note that many attacks have multiple secondary attack effects in addition to their primary effects. For example, the primary effect of the Cross-Site Scripting (XSS) attack is to compromise the integrity of the client-side JavaScript code by injecting scripts. When script injection succeeds, the injected script allows an attacker to achieve different types of goals such as stealing sensitive information (i.e., confidentiality) modifying the content of an honest Web server (i.e., integrity), stealing identity information, stealthily issuing commands to circumvent authorization, or opening many browser windows to prevent the honest user from using the browser normally (i.e., a DoS attack on the user).

Table 2.1: Summary of Threat Classes

Threat Class	Description	Threat Paths
T1	Threats against the communication channels	L1
T2	Threats against servers	L2
T3	Threats against users by an attacker's server	L3
T4	Client-side injection threats (injecting malicious content in the same-origin sandbox of a trusted server)	L4, L5
T5	Threats against servers via an honest user's Web browser (attacks are possible even without injection into the same-origin sandbox)	L3, L4, L5 → L6
T6	Threats against user computers by arbitrary malicious content	L3, L4, L5 → L7
T7	Threats against local networks	L3, L4, L5 → L8

2.3 Threat Classes

This section defines seven different major threat classes as well as their sub classes. Each of major classes is characterized by the paths as illustrated in Table 2.1.

2.3.1 T1. Threats against Communication Channels

Threats in this class correspond to the threat path L1 in Figure 2.2.

In general, Web applications that are not protected by SSL are at risk from this type of threat. However, even when a Web application uses SSL, there are possible vulnerabilities that allow this type of attack. For example, when a Web application allows SSL and non-SSL access to the same resource, an attacker may trick a user by accessing via the non-SSL channel to steal information.

Threats to Confidentiality

Eavesdropping, or an act of stealing information that is exchanged between an honest user and an honest server over the network, threaten the confidentiality of the messages.

Threats to Integrity

An attacker may compromise or alter the data that is transmitted over the network to attack the user or a server.

For example, if an HTTP response is not protected from unauthorized modification, an attacker may modify the content to insert arbitrary script code to perform a Cross-Site Scripting (XSS) attack (See Section 2.4.3).

SSL Stripping refers to attacks in which an attacker rewrites the HTML content in HTTP responses to replace `https://` URLs with similar looking URLs, to trick the user into believing that the URLs being accessed belong to an honest server while they actually belong to the attacker.

2.3.2 T2. Threats against Servers

Attacks in this class correspond to the threat path L2 in Figure 2.2.

Threats to Confidentiality

Examples of threats against confidentiality in this category are:

- **Improper information disclosure:** Vulnerabilities often lead to improper information disclosure. For example, improper error handling often discloses the server's internal configuration information, such as the names and the versions of the software.

- **Directory indexing:** This vulnerability may allow listing all of the files in a directory, allowing an attacker to learn about a structure of the Web server's local files, knowledge which could be leveraged in subsequent attacks. Directory indexing vulnerabilities often occur due to misconfigurations of Web application servers.

Threats to Integrity

Examples of threats against integrity in this category are:

- **Injection attacks:** Many attacks on the integrity of servers aim at injecting malicious scripts or code via an HTTP request to be executed on the server side, such as SQL injection, LDAP injection, or OS shell command injection. The malicious code can be injected through HTTP request parameters, HTTP cookie values (a.k.a. Cookie Poisoning), or the content in the body of a request message (such as XML or JSON data).

- **Shellcode:** Buffer overflow is one type of attack against server integrity but at a lower level than injection attacks such as SQL injections. For example, a format string attack takes advantages of the intrinsic vulnerability of the `printf` function in C to overwrite the stack frame and execute shellcode.

- **Remote File Inclusion:** Instead of injecting the script itself, an attacker injects the location of a remote script file (such as a PHP source file) in a way that it will be imported and executed by a vulnerable server-side Web application.

- **Link flooding:** This is a type of integrity threat but does not necessarily execute malicious code on the server. The objective of link flooding is to insert many hyperlinks to a web site, often to increase the search engine ranking of the site, or to trick users into visiting the site (such as a phishing or pornography site.)

Threats to Authenticity

In this class of threat, an attacker aims at impersonating an honest user by stealing or predicting authentication information or an authorization credential or by exploiting various server-side vulnerabilities. Examples of threats against identities in this category are:

- **Session Hijacking:** Session hijacking is the act of impersonating an honest user by stealing the authorized session information, such as a session identifier stored in an HTTP cookie.

- **Session Fixation:** Instead of stealing the session information, an attacker may create a session identifier, and trick an honest user into using the session identifier. The attacker can then impersonate the honest user by using the session ID.

- **Credential Prediction:** Impersonates an honest user by guessing a session identifier. This threat works against a server that uses predictable session identifiers.

- **Brute Force:** Brute force is a type of attack that tries to discover a secret value (such as a cryptographic key) by trying a large number of possible values. In the context of Web application security, brute force

attacks can be used to discover log-in credentials or a session identifiers to impersonate users.

Threats to Authorization

In this class of threats, an attacker seeks to circumvent the user authorization process and to perform privileged operations on an honest server.

- **Insecure Direct Object References:** Insecure direct object reference occurs when a Web server takes a reference to an object as a request parameter, and allows a requester to access the object without a proper authorization check. For example, some Web applications take an object ID as a request parameter, and use the ID to query databases. An attacker may gain unauthorized access to the object by directly specifying the ID in a request.

- **Directory Traversal (Path Traversal):** is a form of insecure direct object reference that targets the file system. Often this threat occurs due to a vulnerability in or a misconfiguration of a Web application server that allows access to paths that are outside of the public Web data directory by using a URL such as `http://example.com/foo/../../../bar`.

- **Insufficient access control on URLs:** is a form of insecure direct object reference. This type of threat occurs when a Web server does not check the authorization of each HTTP request, but rather tries to control access to certain URLs by rendering or not rendering the hyperlinks to those URLs in the HTTP documents.

- **Referrer Spoofing:** Referrer Spoofing is a type of attack in which a user sends an incorrect or modified Referrer header in an HTTP request. Some Web servers check the Referrer header to determine whether the request is authorized. An attacker may use Referrer Spoofing to gain unauthorized access to such web sites.

Threats to Availability

In this class of threats, an attacker typically seeks to overloading an honest Web server with many HTTP requests and to make it unresponsive to legitimate requests.

2.3.3 T3. Threats against Users by an Attacker's Server

Threats in this class correspond to the threat path L3 in Figure 2.2. This section focuses on scenarios in which an attacker's server deals directly with honest users.

Threats to Authenticity

These attacks can be used to trick a user into believing an attacker's server is an honest server. These attacks can be considered as threats against the identities of servers. These attacks can be used, as in phishing or similar techniques, to steal not only user identities but also other sensitive information.

- **Phishing:** In the context of Web application threats, phishing is an attempt to criminally and fraudulently acquire sensitive information,

such as user names, passwords, or credit card details, by masquerading as a trustworthy server. The word phishing comes from "fishing", referring to the schemes used to lure victims with spam messages with links to the attackers websites.

- **Pharming:** Pharming is a kind of passive phishing attack by redirecting a website's traffic to an attacker's website. Techniques for pharming include changing the hosts file on a victim's computer with malware, compromising DNS responses, or by exploitation of vulnerabilities in DNS server software (such as DNS poisoning). Drive-by pharming (Section 2.3.7) is a local network attack that can be used for pharming.

- **Typosquatting:** Typosquatting, also known as URL hijacking, refers to the practice of registering domain names that are typographically similar to popular Web sites. For example, likely misspellings of URLs of popular Web sites are common examples of Typosquatting. Such domain names can be used for phishing attacks.

Threats to Confidentiality

The threats to the confidentiality of user data should not be major problems as long as the users distinguish trusted servers from untrusted ones. However, it is often the case that a user trusts a server with poor security, and this results in undesirable data leakage. For example, when a user is using a hosted SaaS application such as webmail, the user may send sensitive information via e-mail without realizing its sensitivity and that the content of the e-mail is accessible not only to the recipient but also to the SaaS provider. In a more subtle example, a user may use a search engine to search keywords

that characterize some intellectual properties (e.g., patents), which result in the leakage of these keywords to a third-party search engine server.

2.3.4 T4. Client-Side Injection Attacks

Threats in this class correspond to the threat paths L4 and L5 in Figure 2.2.

The immediate objective of the attacks in this class is to render or execute an attacker's malicious data in a user's Web browser, as part of an honest server's webpage. In other words, the immediate objective of this class of attacks is to render or execute an attacker's data *within the same-origin sandbox* of the trusted content of an honest server.

The attacks can occur either by using input data from an attacker as a Web user or as a result of the mashup of an attacker's Web service. In either case, an honest server may try to filter out malicious injection data, but often fails due to difficulty of proper filtering.

Threats to Integrity

The immediate goal of client-side injection attacks is to compromise the behavior of the client-side Web application, as exemplified by the Cross-Site Scripting (XSS). Note that once the client-side Web application is compromised, this allows the attacker to achieve the primary objective. For example, a malicious script may (1) steal sensitive information such as passwords or credit card numbers as the user enter the data into input fields provided by other applications, (2) modify information in parts of the webpage that appear to belong to other applications to display false or misleading information, (3) redirect the user to a malicious website that is hosting a phishing

attack, or (4) transmit session cookies to the attacker, allowing some authenticated sessions to be hijacked.

There are many variations of XSS that try to circumvent content filtering:

- **Cross-Site Scripting (XSS):** XSS is the most prevalent Web application threat. An attacker tries to inject JavaScript code into the client-side application to execute it in an honest user's Web browser. More details of XSS are described in Section 2.4.3.

- **Cross-Site Image Overlay (XSIO):** XSIO is a variant of XSS which injects a malicious style sheet into the content instead of JavaScript. More details about XSIO appear in Section 2.4.3.

- **Cross-Site Flashing:** Cross-Site Flashing is a variant of XSS which exploits certain vulnerabilities of Flash objects. Some Flash objects take parameters as attributes of the `<object>` or `<embed>` tag which embeds the Flash object into the HTML document, and an attacker can inject script via these attributes.

- **JSON Injection:** JSON is a lightweight format for representing structured objects as a subset of JavaScript. An application may be vulnerable to injection through JSON strings if it does not handle the JSON strings properly. More details are described in Section 2.4.5.

- **Feed Injection:** Feed injection is a type of XSS attacks that misuses XML content feeds such as RSS or Atom. Content feeds can be subscribed to by modern Web browsers, by local Feed readers, or by Web-based services such as Bloglines. The feed reader implementation determines how to interpret any HTML fragments in the body of the

content feed. Some vulnerable browsers do not strip out any special characters (such as < and >) at all, and this allows browsers to execute any script tags in the feed.

- **jar protocol XSS:** the `jar:` is one of the URL protocols supported by major browsers. For example, the URL `jar:https://example.com/a.jar!/b.html` will be interpreted by a Web browser as the `/b.html` file in the JAR file at `https://example.com/a.jar`. The file `b.html` will be executed within the same-origin sandbox as the `example.com` domain. Some types of Web servers, such as Web mail or collaboration systems, allow users to upload JAR files as attachments. By uploading malicious content as part of a JAR file, an attacker can perform XSS attacks. In addition, since the JAR format is the same as a ZIP file, any ZIP files on ZIP-based file formats (such as Open Document Format (ODF)) can also be used.

- **GIFAR:** GIFAR is a hybrid file type that can be created by concatenating a GIF file and a JAR file. Surprisingly, most Web browsers will render a GIFAR file as an image, while the file can be also accessed as a JAR file. GIFARs increase the threats of JAR protocol XSS, making it more difficult to detect attacks because the GIFAR file is disguised as a static image file.

- **HTTP Response Splitting:** The attacker sends a single HTTP request that forces the Web server to create an output stream, which is then interpreted by the victim Web browser as two HTTP responses instead of one response. The problem is that the second response is

under the full control of the attacker, so that the HTTP status line, all of the HTTP headers, or even the response body. The attack can be used to inject scripts into the response body.

- **Script fragmentation:** Script fragmentation is a type of attack that breaks Web exploit messages into smaller pieces so that attack detection by signature analysis becomes more difficult, especially when the protection mechanisms inspect the content at the packet level. This is similar to the TCP fragmentation attacks.

Threats to Authenticity

There are some variants of XSS that focus on impersonating a legitimate user by extending an XSS attack.

The currently most prevalent use of the client-side injection attack is to steal user session information that is stored in the HTTP cookies. It is quite easy for JavaScript code to read cookie values via the built-in `document.cookie` variable (See Section 2.4.3 for an example.) However, in modern browsers, HTTP cookies can be protected by the `httpOnly` attribute in the `Set-Cookie` HTTP response. When the `httpOnly` attribute is specified, a comformant browser prevents the client-side JavaScript code from reading the cookie values.

The following variants focus on compromising HTTP cookies even when they are protected by the `httpOnly` attribute.

- **Cross-Site Tracing:** `TRACE` is a type of HTTP request method that lets the Web server respond with the content of the HTTP request, similar to the echo command. An injected JavaScript code could issue

an HTTP TRACE command to the target server, and then read the cookie values from the HTTP request headers in the echo-response.

- **Cross-Site Cooking:** Cross-Site Cooking allows an attacker's website to send arbitrary cookies to the target Web server via an honest user's Web browser. This is possible by relaxing the cookie domain in the set-cookie header to the super-domain of the target domain. Cross-Site Cooking can be used for session fixation attacks to force the user to continue using a session identifier controlled by the attacker.

2.3.5 T5. Threats against Servers via an Honest User's Web Browser

Attacks in this class occur when an honest user browses some malicious content provided by an attacker. The malicious content does not have to be rendered or executed within the same-origin sandbox of the trusted content. Therefore, the attacks in this class may occur whenever users can be exposed to an attacker's content. In this threat class, the malicious client-side Web applications can be transferred through the paths L3, L4, and L5, and then the content may attack honest servers through the path L6.the user's local computer thorough the paths L7.

Threats to the Authorization

Threats to the authorization in this class are charactersized by the Cross-Site Request Forgery (CSRF) attacks. CSRF (pronounced as "sea-surf", sometimes written as XSRF) is a type of attack that tricks a user into issuing commands to a Web server without knowing anything is happening. The

attack works by including a link or script in a page that accesses a site to which the user has been authorized. Unlike XSS, CSRF does not require malicious script to be injected into the webpage. More details of CSRF are described in Section 2.4.6.

Threats to Confidentiality

Threats to the confidentiality include stealing informatoin on an honest server, via an honest user's browser, without injecting script or content into the same-origin sandbox of the target server domain. JavaScript Hijacking and CSSXSS are examples of such attacks. These attacks can be seen as variants of CSRF, because when the resources on the target server are protected, these requests can be authorized as a result of the CSRF vulnerabilities.

- **JavaScript Hijacking:** JavaScript hijacking is a variant of CSRF that allows an attacker to read sensitive data from authorized sessions, via a legitimate user's browser. Section 2.4.6 describes more details of JavaScript Hijacking.

- **Cross-Domain Information Disclosure Vulnerability (CSSXSS):** A Cascading Style Sheet (CSS) can import arbitrary files from any domain by using the `@import` directive. Some old versions of IE allow importing files even if their content types are not CSS [92]. CSSXSS can be used to steal information from an honest server from client-side Web applications running in a user's Web browser. Although it is called CSSXSS, the attack does not require injection of malicious content into the trusted website. An attack can occur when a user browses the attacker's content on any web site.

2.3.6 T6. Threats against User Computers by Arbitrary Malicious Content

The targets of this class of threats are the personal computers and Web browsers of honest users.

In this threat class, the malicious client-side Web applications can be transferred through the paths L3, L4, and L5, and then the content may attack the user's local computer thorough the path L7. That is, the threat appears after the a user browses the malicious content, which is hosted on either an attacker's Web server or an honest Web server. It is not necessary to render or execute the attacker's content in some same-origin sandbox of the trusted content.

Threats to Confidentiality

An example of a threat to confidentiality is the theft of the browser history from the stylesheet attributes.

Stylesheets can be used to change the style of hyperlinks in a webpage based on the browser history. For example, a stylesheet can specify different colors of links based on whether or not the user has visited the link. By exploiting this feature, an attacker can capture a user's browsing history by reading the current style from the browser DOM in the JavaScript code. Online advertisements often use this technique to access a user's browsing history, and then show personalized advertisements for each user. In addition, it is theoretically possible to steal the search query history by using the same technique [51].

Threats to Integrity

The attacks in this class can be used to compromise the integrity of the user's computing environment. Examples of such attacks include:

- **Cross-Zone Scripting:** Cross-zone scripting is an attack which takes advantage of a vulnerability in a zone-based browser security model. The attack allows a script in unprivileged zones (such as the Internet zone) to be executed with the permissions of a privileged zone (such as the local computer zone). An example of this type of attack is to let the Web browser download and cache a JavaScript file, possibly masquerading as another content type (such as an image file), and then execute the file directly from the local cache with the local file priveldge.

- **Clickjacking:** Clickjacking is an attack that tricks a user into believing that a clicked link is something different from what it is. The attack can be carried out by overlaying transparent objects over the buttons or links that user wants to click, so that when a user appears to click on a button, the actual click is on the overlaid object [49].

- **Malicious Iframes:** *Malicious iframes* (also known as *iframe injection*) is a type of attack that tries to run malicious code on the target user's Web browser by injecting an iframe that points to an attacker's website. The attack became infamous when several Italian websites were hacked with inserted iframes, which exploited a browser vulnerability to install Torojan code in users' computers [39][44].

- **URI Protocol Handler:** A desktop application on a user computer

can register specific URL handlers with the Windows registry so that the application is invoked when a special URI is incuded in a webpage. A vulnerability in the Firefox browser allowed an attacker to invoke Firefox with a `-chrome` option by using a special `firefoxurl:` URI protocol handler. When invoked with this option, the attacker's code could access local files and other OS-level resources [72].

Threats to Availability

A *browser crasher* is a type of DoS attack against user computers. Typical techniques include opening an infinite numbers of browser windows or creating frames from JavaScript code in a webpage.

2.3.7 T7. Threats against Local Networks

In this threat class, the malicious client-side Web applications can be transferred through the paths L3, L4, and L5, and then the content may attack neighbouring servers and devices through the path L8, by using the user's browser as a stepping stone. This class can be seen as a subclass of T5 with the slight difference that the targets of the attacks are devices and servers within the user's local network. In most cases, the client-side JavaScript code is executed on a user's Web browser that runs inside of a firewall, and thus the technique can attack servers and devices within an intranet.

Attacks in this class include:

- **Port Scanning:** JavaScript-based port scannig allows an attacker to issue HTTP requests by dynamically generating HTML elements and

attributes, and thus detect the presence of devices or information such as the type of Web server software.

- **Cross-Site Printing:** Cross-Site Printing is a type of attack that uses JavaScript and HTML forms to send print commands to network printers.

- **Drive-by Pharming:** A drive-by pharming [86] attack allows an attacker to override the DNS setting in home broadband routers by using JavaScript code.

More details of these attacks are described in Section 2.4.7.

2.4 Descriptions of Major Threats

This section gives details of the selected major threats that are relevant to foci of this thesis work.

2.4.1 Server-Side Injection Attacks

Injection attacks are some of most prevalent threats to server-side applications. An immediate focus of an injection attack is to compromise the application's behavior, which can then be used to also compromise confidentiality.

A server-side injection attack occurs when an attacker sends crafted data to a Web application, and the Web application uses the data in a way such that the data is interpreted as part of commands or queries, rather than as static data value. A typical injection attack is SQL injection, which allows an

attacker to execute an SQL query that is not intentionally permitted for the application. Similar threats are observed with LDAP queries, XPath queries, and OS commands.

For example, Figure 2.3 is an example of a server-side Web application with an SQL injection vulnerability.

```
String username, query;
username = request.getParameter("name" );
query = "SELECT * FROM users WHERE name = '" + username + "';" ;
Statement statement = con.createStatement();
ResultSet results = statement.executeQuery(query);
```

Figure 2.3: Example of a Web Application with SQL Injection Vulnerability

In this example, the value of the `username` variable is taken from an HTTP request parameter, and the value is used in an SQL query without any prior check or sanitization. The SQL query is intended to retrieve the database entries from the table `users`, where the value of the `name` column matches the given `username`.

An expected value for the `username` is an ordinary user name for an entry in the database, such as `John Smith`. However, an attacker may use a crafted string such as

`' or '0'='0`

as the name. As a result, the SQL query string may become

`SELECT * FROM users WHERE name = '' or '0'='0';`

Because of the effect of the condition `name='' or 0=0`, the query matches all the entries in the `users` table and returns the entire content to the attacker, resulting in a massive data leakage incident.

2.4.2 Information Leakage from Servers

As described in Section 2.3.2, improper information disclosure is one of the most prevalent causes of confidentiality threats.

Server-side injection attacks can result in improper information disclosure as described in Section 2.4.1. Other typical leakages occur because of mistakes by Web application developers. For example, when an application error is not handled properly, a server may return error messages to the client, which include the details of internal information about the server (such as Java stack traces caused by exceptions). It is also a common problem that the server-side Web application code output sensitive information (such as user passwords) into log files, and this can result in the leakage of such information.

In addition, some attacks against authorization can result in compromise of data confidentiality. For instance, Insecure Direct Object References is a typical attack to circumvent authorization checks by Web application. It occurs when a Web server accepts a reference to an object as a request parameter, but allows a requester to access the object without a proper authorization check.

For example, assume that a Web application expects HTTP requests such as:

```
http://abc.com/showFile?filename=xyz.txt
```

In the above URL, the `filename` parameter includes the name of a file in a certain directory, from which the Web application reads and returns the content. However, an attacker may request a crafted URL such as

```
http://abc.com/showFile?filename=../../../etc/passwd
```

In which case, the content of `/etc/passwd` could be returned if no appropriate countermeasures are installed.

2.4.3 Client-Side Injection Attacks

Cross-Site Scripting (XSS) (Section 2.3.4) has been one of the most frequent attacks since early days of Web 1.0, and it has been reported that XSS is 70% of all Web application vulnerabilities [42]. XSS can be categorized into three types: *stored-XSS*, *reflected-XSS*, or *DOM-based XSS*.

Stored XSS

In *stored-XSS*, the injected script is first stored in persistent storage (such as a relational database) and later is injected into HTML. A typical example of stored-XSS would occur in an online discussion board or blog comments, when a message from a malicious user is presented to other users without sanitization.

Reflected XSS

The *reflected-XSS* occurs in an application that *reflects back* user input data without sanitizaiton. A typical example is a search engine that shows the search keyword along with the search results. An example of a URL which carries out a script injection attack might appear as:

```
http://trusted.com/search?keyword=xss<script>document.images[0].src
="http://evil.com/steal?cookie=" + document.cookie; </script>
```

When a user accesses this URL, the user will be brought to the search result page of `trusted.com`, but instead of showing the real search results,

the script element inserted in the URL will be executed. As a result, the user's document cookie for `trusted.com` will be sent to the attacker's server at `evil.com`.

DOM-based XSS

The third type of XSS, *DOM-based XSS*, is receiving increased attention these days [63]. The DOM-based XSS vulnerability is possible when the client-side application is designed to read a URL string from `document.location`, `document.URL`, or `document.referrer`, and inject part of it into the DOM [63].

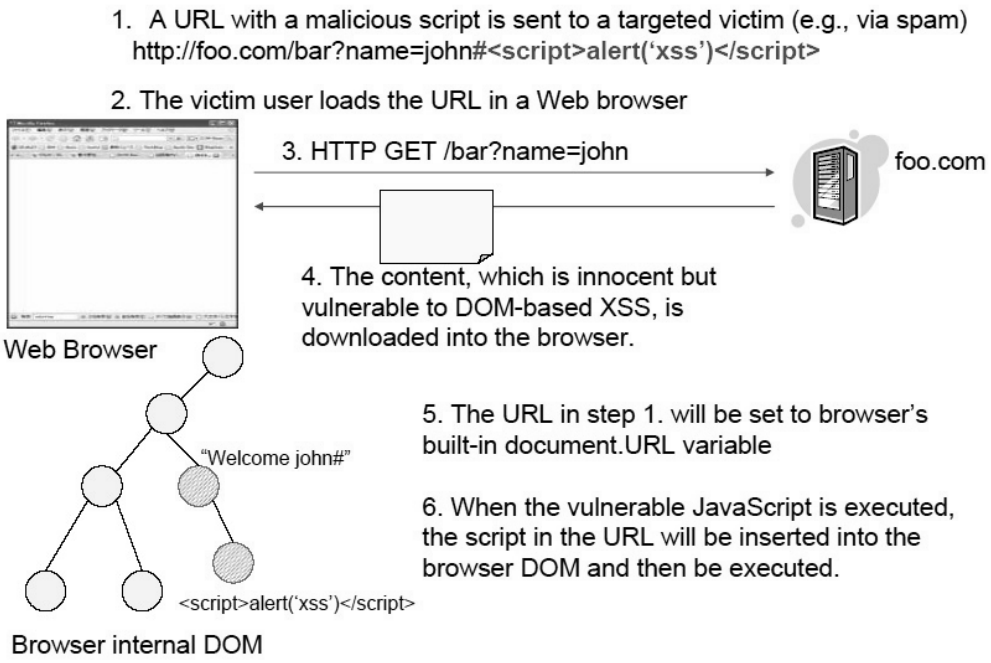


Figure 2.4: An Example of DOM-based XSS

Figure 2.4 shows an example of the DOM-based XSS. An example of

vulnerable client-side JavaScript is shown in Figure 2.5. The attack can be carried out in these steps:

1. The user receives a malicious URL, such as `http://foo.com/bar?name=john#<script>...` `...</script>` via a channel outside of the browser, such as in spam e-mail.

2. When the URL is passed to the browser, e.g., by the user clicking on the link in the spam mail, the browser sends the HTTP request `GET /bar?name=john` to the server foo.com. The fragment identifier after the `#` sign will not be sent to the server.

3. The server `foo.com` returns an HTML document that includes vulnerable JavaScript in Figure 2.5.

4. The URL of the document, including the fragment identifier, will be assigned to the browser's built in variables such as `document.location`.

5. The returned HTML document is parsed and rendered by the browser, and the embedded JavaScript code is executed. The vulnerable JavaScript code in Figure 2.5 tries to retrieve the `name` parameter in the URL, and then print it on the page. However, since the value of the variable `s` will include the entire string after `name=`, not only "Welcome john" but also the following script tag will be inserted into the document. As a result, the script will be executed.

This attack cannot be prevented by the same-origin policy, because the URL string is not explicitly associated with any origin but implicitly trusted.

```
<html><body> ..
 <script>
 var pos=window.location.indexOf("name=")+5;
 var s = window.location.substring(pos,
                  document.location.length);
 document.write(“ Welcome “  + s);
</script>..
</body></html>
```

Figure 2.5: Example JavaScript vulnerable to DOM-Based XSS

It is implicitly regarded as in the same origin as the HTML document, even if it comes from an untrustworthy source.

Once a malicious script is triggered, it is executed within the *same-origin* sandbox of the innocent content, and therefore allows an attacker to steal document cookies or user passwords, or compromise data in a Web application for such purposes as phishing [89]. Likewise, mashup applications have the same vulnerabilities as XSS, because third-party content is often integrated and executed in the same-origin context as trusted content.

XSS Obfuscation Techniques

A common practice to prevent XSS is to filter out scripts on the server side. However, for DOM-based XSS, server-side filtering is not possible because the malicious URLs are handled only on the browser side. Even when the server side filtering is logically possible, it is difficult to correctly filter data because new ways of bypassing filters are constantly being devised [45][12].

Figure 2.6 shows examples of XSS obfuscation techniques. All six examples in Figure 2.6 result in execution of the same script.

It is also a common practice among attackers to use the URL redirec-

```
1: <script>alert('mal');</script>
2: <img src=“ javascript:alert('mal'); ” />
3: <img src=“ jav
          ascr
          ipt:alert(‘ mal');“  />
4: <img src="J%97vasc%72ipt%3Aalert%28%27mal%27%29%3B ”/>
5: <img src=“ &#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;
   &#116;&#58;&#97;&#108;&#101;&#114;&#116;⋯ />
6: <meta http-equiv=“ refresh ” CONTENT="0;url= data:text/html;
   base64,amF2YXNjcmlwdDphbGVydCgnbWFsJyk7“  />
```

Figure 2.6: Examples of XSS Obfuscations

tion function of Web servers to hide the malicious URLs so they will not be detected by the user. For exapmle, TinyURL.com provides a service to generate a short alias URL (such as http://tinyurl.com/5hmy6j) for an arbitrary URL, and then redirects the user to the original URL when accessed. An attacker may send the short alias in a spam message, decreasing the chance of detection even by careful users.

Stylesheets and Cross-Site Image Overlaying (XSIO)

Cascading Style Sheet (CSS) is a stylesheet technology widely used to control the presentation of Web content. JavaScript used to be the only type of active content that could carry out an XSS attack. However, due to rich capability of CSS, the most prevalent style sheet technology used in the Web, there are some attack variations that are done by injecting stylesheet.

A CSS stylesheet can be embedded in an HTML element by using `<style>` elements, or imported from a given URL by using `<style src=‘‘...’’>`. In addition, each HTML element may have a `style` attribute to specify the presentation of the content in the element.

With Ajax applications, the client-side JavaScript code may change the CSS in the DOM to dynamically control the presentation of the webpage.

There are some possible XSS attacks via CSS. For example, the `background` properties of CSS allows specifying the URL of a background image to be rendered on the webpage. However, when a pseudo-URL that starts with `javascript:` is specified, the following string will be executed as JavaScript.

Some implementation-specific problem also complecate the problems. For example, Internet Explorer supports an `expression` function in CSS which can executes arbitrary JavaScript code with this format:

```
<div style="left:\expression( alert('xss') )">
```

Note that some browsers handle CSS strings in a manner different from HTML, which can make the detection of attacks difficult. For example, IE allows different character escapes in CSS, and some multi-byte characters are recognized as their equivalent ASCII characters. Therefore, a content filter that only looks for the string `expression` will fail to detect an obfuscated string such as:

```
<div style="left:\0065xp R ession( alert('xss') )">
```

where `\0065` is the hexiadecimal representation of ‘‘`e`’’ (U+0065) and ‘‘ R ’’ (U+0280) is a full-width equivalent of ‘‘`R`’’ (U+0052).

Cross-Site Image Overlaying (XSIO) is an attack that succeeds even without executing JavaScript from CSS [90]. For example, an attacker may insert an inline image in the user feedback of an online shopping site, and then by using the style attribute, the attacker may position the image on top of the logo image of the online shop. The attack becomes more effective when the attacker uses a hyperlink with the image, so that the victim who clicks on

the logo will actually be directed to an online phishing site.

2.4.4 Mashup Security

Mashup involves techniques for Web applications that integrate data or functions from different sources to create a new service. An example of a mashup application is a Web-based restaurant guide that integrates an online map service to show the locations of the restaurants combined with blog entries that provide reviews of those restaurants. The mashup approach allows creating new applications by leveraging content and services in existing websites.

However, a mashup application may pose new security risks, because a typical mashup application integrates content and services from multiple servers with different trust levels into a single Web page. Hence if any of the sources has a malicious intention, that source might inject malicious client-side JavaScript code to attack the applications from the other websites. This is almost identical to the Cross-Site Scripting (XSS) (See Section 2.4.3) attack except that the attacker is a malicious Web server rather than a Web user.

Most of today's Web browsers use the same-origin policy as a de facto standard security model, and therefore try to prohibit suspicious communication between browser windows and frames associated with different domains. However, the mashup of contents and services from different domains often calls for data exchanges, and many mashup applications wind up trying to bypass or evade the same-origin policy by using various server-side or client-side techniques.

The server-side mashup method aggregates content from several third parties on the server of the mashup Web application so that the user's Web

browser regards all of the content as coming from a single domain, and the communication between the source domains is actually taking place in the server.

In contrast, the client-side mashup method uses various loopholes in the same-origin policy while creating the mashup application in the user 's Web browser. A typical client-side mashup might use `<script>` tags to import client-side JavaScript code for third-party services. Since the same-origin policy is applied only at the granularity of the HTML document loaded within a browser window or frame, the imported JavaScript is regarded as belonging to the same domain as the HTML code that imported it, and this allows the effective aggregation of content from different domains. In addition, though the same-origin policy can be applied to an asynchronous network access by using the XMLHttpRequest API, it is not applied to the network accesses originating with `<img>` or `<script>` tags in the HTML, and thus the client-side JavaScript code can also communicate with third-party servers by dynamically modifying the src attributes of these elements.

Once the content from different domains is rendered in the same browser window or frame, it is not possible with the current browser architecture to enforce access controls. In today's browser architecture, all of the JavaScript within a single window or frame shares the same execution context, and cannot limit the access to variables or functions. In addition, existing variables or functions can easily be overwritten if a variable or function with the same name is declared. Any content within a browser window or frame can access the browser DOM (Document Object Model) tree for that window or frame and read or write information that may belong to other applications.

Today's mashup applications assume there are trusted relationships between the mashup providers and the service providers to maintain the security of the application. However, even if a particular third-party service provider is not malicious, it may be vulnerable to XSS or other attacks, and other mashup applications can then be at risk through the vulnerable service. In particular, the screen‘ real estate ’used by advertisements is often subleased to other providers, making the trustworthiness of the actual content provider unclear.

2.4.5 JSON

JavaScript Object Notation (JSON) is a text-based, lightweight data description format widely used in Ajax applications. In JSON, an object is represented in name-value pairs, and a JSON string can represent a nested structure of objects and arrays.

Although JSON is independent of any programming language, it has high affinity with Web applications, because its syntax is nearly identical to a subset of JavaScript.[1] When a JSON string is evaluated as a JavaScript string by using the `eval()` function, the data is transformed into a JavaScript object, which is a handy technique often used in today's Web applications.

However, the use of `eval()` for parsing JSON strings is dangerous, because if any JavaScript is included in a JSON string, then the script will be executed. (See Figure 2.7 for an example of JSON.)

[1]Precisely speaking, the JSON syntax defined in RFC 4627 is not a subset of JavaScript. In particular, each object property name has to be a string in RFC 4627 (e.g., { 'name': 'value' }), while it is a name token in JavaScript (e.g., { name : 'value'}). However, as a typical practice, many existing Web applications use the JavaScript syntax to exchange structured data between Web servers and clients

```
01: var jsonstr = “ [ { name: ’Sachiko Yoshihama’,
    affiliation: ‘ Yokohama National University’,
    destination: ’Barcelona’, date: ‘ March 25, 2010’ },
    { name: ‘ Tsutomu Matsumoto’,
    affiliation: ‘ Yokohama National University’,
    destination: ’New York’, date: ‘ February 28 2010’} ] ”;

  : // Evaluate the JSON string with eval()
  : // to convert it into a JavaScript object
02: var obj = eval(jsonstr);

  : // “ Sachiko Yoshihama ” is rendered
03: document.write(obj[0].name);
```

Figure 2.7: Example of JSON

It is necessary to check the validity of a JSON string (e.g., by using the regular expression recommended in RFC 4627 [28]) before calling `eval()`. An even safer solution is to implement a JSON parser in JavaScript to avoid the use of `eval()`.

Since JSON is just plain text with a simple bracketed structure, JSON messages can be exchanged via many channels. In particular, JSON messages are used to support cross-domain communication in mashup applications. Because of the same-origin policy, the XMLHttpRequest API cannot be used to communicate with external servers. JSON with Padding (JSONP) is a well known technique to bypass the same-origin policy by using JSON in combination with the `<script>` tag, as shown in Figure 2.8.

When JavaScript code dynamically inserts a `<script>` element, the browser accesses the URL in the `src` attribute. This results in sending the information in the query string to the server. In Figure 2.8, the username and reservation

```
<script type="text/javascript"
  src="http://travel.com/findItinerary?username=sachiko&
  reservationNum=1234&output=json&callback=showItinerary" />
```

(a) `<script>` tag which initiates a GET request to retrieve data

```
showItinerary( { name: 'Sachiko Yoshihama',
    affiliation: ' Yokohama National University',
    destination: 'Barcelona', date: ' March 25, 2010' } );
```

(b) A JSON object in the HTTP response message is returned via a callback function

Figure 2.8: Example of JSONP

are passed as name-value pairs. In addition, the query string contains the output format requested by the client and the name of the callback function (`showItinerary`). When the `<script>` tag is loaded, the callback function is executed, and the information returned from the server is passed to it through its arguments.

2.4.6 Cross-Site Request Forgeries (CSRF)

Cross-Site Request Forgery (CSRF) is a relatively new class of attacks that is receiving attention in recent years. CSRF represents threats in the class T5 (See Secton 2.3.5) and takes advantage of a very generic and widespread vulnerability in the session management of Web applications. In general, a website that authenticates and authorizes a user will return an authorization cookie to the browser after a legitimate user logs in. The Web browser stores the cookie until its expiration, and when the browser access the same website the next time, it will automatically attach the authorization cookie to the HTTP requests to the site. This mechanism allows the user to avoid entering

the password for every request, while still allowing the website to recognize HTTP requests from authorized users.

However, this generic mechanism has an obvious weakness in that the authorization cookie will be sent to the website everytime the browser issues HTTP requests, whether or not the user is aware of such authorization requests. For example, let's assume that a legitimate user logs into an online shopping site. Before the authorization cookie of the shopping site expires, the user may visit another website that includes malicious JavaScript code. If the malicious JavaScript code issues HTTP requests, such as a purchase request, to the online shopping sites, then this request will be accepted by the online shopping site as an authorized request, since the authorization tokens are attached to the HTTP request header.

CSRF is also a valid attack for authorization mechanisms other than cookies. For example, HTTP authorization and SSL client authentication have the same weakness.

An example of a major CSRF insident was the "Hamachi-chan" attacks, which posted multiple messages by using many different user accounts in a social networking site in Japan in 2005 [36]. Any HTTP-based API has potential vulnerablilities to CSRF.

Unlike XSS, CSRF does not require the malicious script to be injected into a legitimate website. Therefore, many websites have potential CSRF vulnerabilities.

JavaScript Hijacking

JavaScript hijacking is a variant of CSRF, which allows an attacker to read sensitive data from authorized sessions via a legitimate user's browser using a technique similar to JSONP [26] (See Section 2.4.5 for more details about JSON and JSONP).

In JavaScript hijacking, an attacker overrides the built-in methods of JavaScript objects to steal data as a JSON string is being converted into an object.

Figure 2.9 shows an example of a JavaScript hijacking attack:

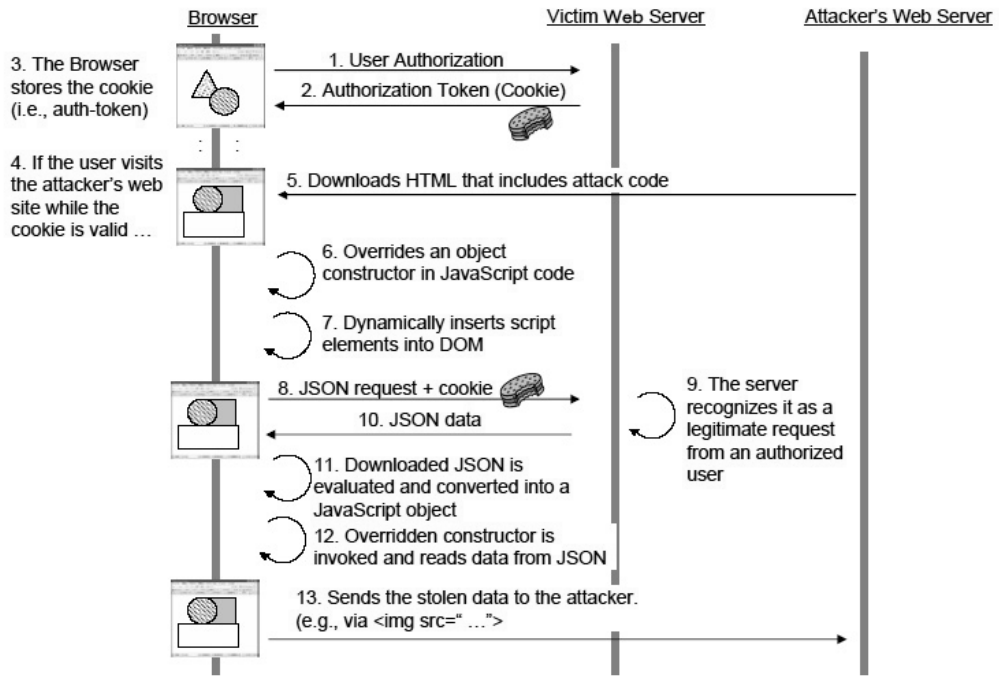


Figure 2.9: An Example of a JavaScript Hijacking Attack

1. An honest user authenticates and authorize s a session with an honest

Web server.

2. The honest Web server returns authorization tokens, typically as HTTP cookies.

3. The Web browser of the user stores the cookies and associates them with the URL of the honest server.

4. Now the user visits an attacker's website before the cookies in step 3 expire.

5. The Web browser downloads an HTML file, including malicious JavaScript code, from the attacker's website.

6. The malicious JavaScript code overrides the object contructor (or setter method)

7. The malicious JavaScript then creates a new `<script>` element dynamically, and sets the honest server's URL for JSON data to its `src` attribute.

8. The Web browser issues an HTTP GET request when the `<script>` element is inserted into the DOM. Since the honest server's URL is associated with the authorization cookies, the browser automatically attaches the cookies to the HTTP request. (This step is the same as the CSRF attacks.)

9. When the JSON data is returned from the honest server, the data is evaluated as JavaScript because its download was triggered by an insertion of the `<script>` element.

10. The overridden constructor (or setter) is invoked, allowing the attacker's code to steal object property values while the object is instantiated.

11. The overridden code can send the stolen data to the attacker, perhaps by inserting an `<img>` element with the stolen data as its request parameters.

2.4.7 Local Network Attacks

As described in Section 2.4.5, client-side JavaScript code can easily access any server by dynamically updating the HTML via the browser DOM tree. Local Network Attacks are a class of attacks that uses a user's browser as a stepping stone to attack neighboring servers and devices. In most cases, the client-side JavaScript code is executed by the user's Web browser which runs inside of firewalls, and thus the technique makes it easy to attack servers and devices in an intranet.

Port Scanning

Probing is often a very important first step for an attacker to identify attack targets, by detecting active IP addresses and port numbers as well as the types and versions of the software running on them. JavaScript-based port scannig allows an attacker to issue HTTP requests by dynamically generating HTML elements and attributes with JavaScript. By then observing the access results and the type of errors, the JavaScript code can detect the presence of devices and the types of the Web server software running on them.

For example, a simple way to issue an HTTP request is to generate an `<img>` element dynamically, and set an arbitrary IP address and a port number in the URL in the `src` attribute. When the port is active and the image at the URL is present, the image will be loaded by the browser and will invoke the `onload` event handler. The attacker can determine the status of a port because the request will timeout if the port is deactivated, while it will get an explicit error if the port is active but the image does not exist. An attacker can also try to access default icon images in popular Web server software to determine the types of the servers from the presence of icons.

Cross-Site Printing

In some network printers that do not authenticate users, it is possible to print a document by simply sending a text-based print command to the printer port. Cross-Site Printing is a type of attack that uses JavaScript and HTML forms to send print commands to network printers [94]. The attack can waste massive amounts of paper and ink by printing large amount of data, or print advertisements or spam messages on a victim's printer.

Drive-by Pharming

Pharming is a coined term for an advanced form of phishing. *Phishing* is an analogy of fishing to catch a specific target, while *pharming* is an analogy of farming to harvest crops by casting seeds. In practice, it usually refers to the type of attack in which an attacker compromises DNS reponses to guide a target user to a fake website. Drive-by pharming [86] is a variant of the local network attacks, in which an attacker overrides the DNS settings in home

broadband routers by using JavaScript code. Although home broadband routers usually reuquire the administrator passwords before changing the configuration, it is often easy to predict the password especially when it has not been changed from the default password. If any session information is left in the user's Web browser, an attacker may free-ride on authorized sessions by using CSRF.

2.5 Foci and Non-Foci of This Thesis

Among all of the Web application threats described in Section 2.3, this thesis focuses on the threats that are especially relevant to data security and information flow. This section describes the focus of this thesis.

2.5.1 Foci of this Thesis

This section gives an overview of the contributions of this thesis that are summarized in Table 2.2. More detailed discussion will be provided in Chapters 4, 5, and 6.

Secure Browser Model

The Secure Browser Model (Chapter 4) is a proposal for a new browser security model to protect Web browsers from being used for attacks. In particular, it mitigates the threats T4, T5, and T7:

- **T4. The Client-Side Injection Attacks:** The Secure Browser Model mitigates the risk of the client-side injection attacks by fine grained access control on the Web browsers. The proposed method

Table 2.2: Foci of this Thesis

Theme	Approach	Effects
1. Secure Browser Model (Chapter 4)	A novel access control model for Web browsers that uses the origin of data/content to make access control decisions. Security policy is defined by Web app servers and users.	• Prevent attacks via honest users' Web browsers: T4, T5, T7
2. DIFCA (Chapter 5)	Information flow control by run-time monitoring of a server-side Web application, to control the information flow on the server.	• Mitigate confidentiality and integrity threats to a server: T2 • Server-side protection for client-side injection attacks: T4
3. WebDLP (Chapter 6)	Monitoring and analyzing data flow between Web browsers and servers on a proxy.	• Prevent data leakage threat via Web browsers: T3

allows policy-based control over the execution privilege of the content downloaded from each domain.

- The proposed model mitigates the risks from mashups by enforcing fine-grained access control for each part of mashup content in the Web browsers. Because the model takes the content origins into consideration, it can effectively control behavior of the content (such as any JavaScript) that originated from different domains.
- The proposal currently focuses on controlling the privilege of JavaScript code and does not address other problems such as Cross-Site Image Overlay (XSIO) (Section 2.4.3). However, it is possible to extend the model to support such related threats.

- **T5. Threats against Servers via an Honest User's Web Browser:** The proposed model mitigates the risk of Cross-Site Request Forgeries (CSRF) (Section 2.4.6) by controlling the way that the authorization tokens (such as cookies) are attached to HTTP requests. For example, a webpage administrator can specify that only the webpages from a specific server can issue HTTP requests with the authorization tokens from the same server, and thus preventing CSRF attacks from other domains. Since other attacks in this class (such as the JavaScript Hijacking described in Section 2.4.6) are variants of CSRF, those threats can also be mitigated.

- **T7. Threats against Local Networks:** The model mitigates the risk of Local Network Attacks by enforcing fine-grained access control on the network access from the client-side Web applications. The proposed security model allows an user to specify a security policy to protect local resources. For example, a user may define the access control policy such that no external webpages can initiate HTML requests to IP addresses in the local network, not even through dynamic HTML modifications.

DIFCA

DIFCA (Chapter 5) is a server-side technology for tracking information flow within Web applications and for controlling the flow based on the policy. In particular, it can mitigate the threat class T2 and T4:

- **T2. Threats against Servers:**

  - DIFCA is designed to prevent improper information leakage. For example, it can track how sensitive data (such as a credit card number read from a database) flows in a Web application, and prevent improper disclosure of such data based on the policy. Similarly, direct-object-reference attacks can be prevented by enabling security label propagation to the content read from files. For example, when a file is read by the application program, its content is associated with a security label, depending on the location of the file and the pre-defined policies, such as "high-security" for files other than certain directories. The security label is propagated while the program is executed, and HTTP responses will be blocked when high-security data is being rendered on a webpage.
  - DIFCA can also mitigate the server-side injection threats by tracking the propagation of the tainted data in a server-side Web application. It can detect whether non-trustworthy data is used in operations without proper validation or sanitization. In addition, information leakage that results from server-side injections can be prevented. For example, for the example in Section 2.4.1, information flow control policies can be defined for database entries to prevent information leakage by SQL injection attacks.

- **T4. Client-Side Injection Attacks:** DIFCA can provide a server-side protection mechanism for client-side injection attacks. For example, it can be used to prevent XSS attacks on the server-side, by tracking the information flow in the server-side Web application, and make sure that all untrusted input is filtered for safety before being

rendered in a webpage. The basic approach is known and widely used as *taint analysis*, but DIFAC can track actual information flow that may depends on the run-time condition, without needing modification of the existing application code.

WebDLP

WebDLP (Chapter 6) addresses the data leakage issues between a Web browser and servers. In particular, it focuses on data leakage from a user to external servers in the threat class T3 (Threats against Users by an Attacker's Server). Such leakage typically occurs when a user uploads sensitive data to Web servers, either by mistake or due to malicious intent. For example, when a user is using a Web-mail service, and sends sensitive data either as a message body or as attachment, that action will cause undesirable information flow.

2.5.2 Beyond the Scope of this Thesis

User authentication and authorization are outside the scope of this thesis work. All of the proposals in this work assume that user authentication and authorization are properly done, and try to detect undesirable information flow even under such situations. Therefore, all threats related to authentication and authorization issues are not in the scope of this thesis, except for those in the threat class T5 (i.e., the cross-site request forgery) which occurs due to an inadequate security model of some Web browsers.

Since this thesis focuses on application-level information flow issues, middleware-level threats that occur due to the bad design or misconfiguration of the

Web server software are outside the scope of this thesis. For example, directory indexing and directory traversal are attacks are not explicitly addressed by proposals in this thesis because they usually occur because of miconfigurations of Web servers. Likewise, low-level vulnerabilities such as buffer overflow are outside of the scope, because the proposals here focus on the application-level vulnerabilities in high-level programming languages such as Java or JavaScript.

Transport-layer protection using the SSL (Secure Socket Layer) protocol is currently widely used to secure Web applications, and thus the threat class T1 is outside the scope of this thesis. However, the dynamic data flow tracking technology proposed in DIFCA (Chapter 5) can be used to control data flow from a server-side application to secure and non-secure channels, and to make sure that sensitive data is not output to non-secure channels.

Attacks against user computers often exploit vulnerabilities of native software such as Web browsers and plug-ins. Since this thesis focuses on threats to the Web applications rather than to the native software, the threat class T6 is outside the scope of this thesis.

Chapter 3

Related Work

This chapter briefly introduces technologies for mitigating the risks of Web application threats that were described in Chapter 2.

3.1 Server-Side Protection Technologies

To prevent server-side injection attacks in general, application-level detection and filtering is the most basic countermeasure today. However, since the application-level detection is error prone, *taint analysis* is often used to make sure that all user input is sanitized before being inserted into an SQL query or other commands.

Taint analysis can be classified mainly into two types: static analysis and dynamic analysis. Perl's taint mode is an example of dynamic analysis. It is a special execution mode of a Perl interpreter that tracks data flow from untrusted sources (such as user input) and warns when a *tainted* value is used for a sensitive operation (such as an SQL query). A drawback of this mode is that when a value is sanitized for safety, the application developer

has to explicitly call the *untaint* function to indicate that the value is now safe, which is also an error prone step. There are also many static approaches using program analysis techniques. More observations on information flow control techniques are discussed in Section 3.7.

Some types of attacks can be mitigated by attack-specific techniques. For example, a parameterized query is an effective method for preventing SQL injection attacks.

Buffer overflow is often the method of injection attacks on applications written in C. The risk of buffer overflow can be mitigated by language support. For example, the Java language treats each array as an object and the language interpreter detects and prevents access to the memory space beyond the array boundaries.

Secure operating systems are generic technologies that can enable mandatory access control on files and resource on the operating system level. For example, SELinux [70] is a mandatory access control mechanism for the Linux OS, which allows specifying access control policies for each OS-level process to access files, directories, and other resources. There are other secure OS technologies such as LIDS [8] and Tomoyo Linux [13].

Secure OSes can be used to prevent direct object reference attacks (See Section 2.4.2), that often occur because of bugs in server-side Web applications, by enforcing mandatory access control policies. However, secure OSes can provide only coarse grained protection at the granularity of OS processes and resources (such as files). For example, direct object reference attacks may be carried out against objects to which a Web server process has an access privilege, in which case the OS level protection cannot prevent attacks.

For instance, a Web server process may have privilege to access a database table, but some of its content should not be disclosed to the users.

3.1.1 Contributions of this Thesis

DIFCA (See Chapter 5) enables file-grained information flow tracking and control at the granularity of the programming language. It can prevent injection attacks by tracking the information flow to verify that untrusted data is not used in sensitive operations (such as SQL queries) unless sanitized. Similarly, DIFCA can prevent direct object reference attacks. Even when the data object is directly accessed by an attacker, exposure of its content will be prevented by the information flow tracking and control.

Buffer overflow attacks are out of the scope of this thesis. In particular, DIFCA assumes that the program is written in Java, which is resistant to buffer overflow attacks.

3.2 Technologies for Preventing Client-Side Injection Attacks

Unfortunately, there is no panacea for preventing cross-site scripting (XSS) attacks. That is because there are so many varieties of Web application designs so that a countermeasure that is effective in one application is not always effective in other applications.

This section describes the server-side and client-side technologies for preventing mashups and enabling secure mashups.

3.2.1 Server-Side Technologies

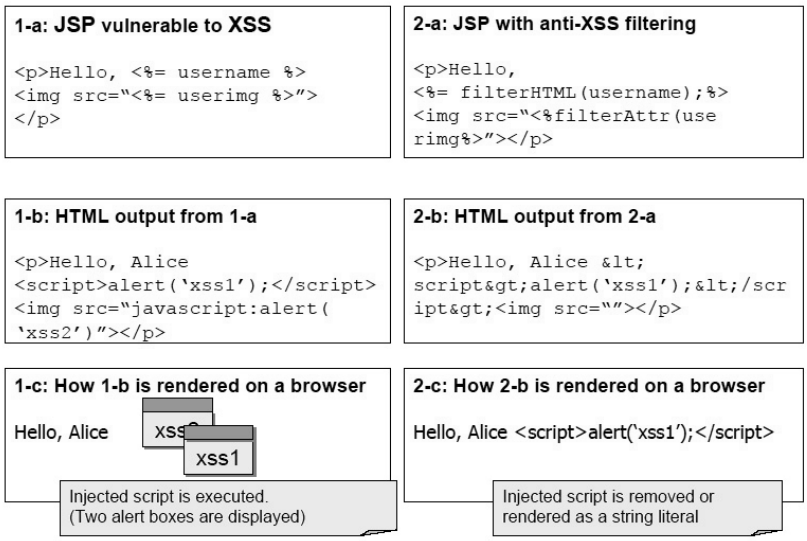


Figure 3.1: Examples of Anti-XSS Filtering on JSP

Content Filtering

The most widely used server-side countermeasure today is to filter or escape user input values to disable any injected content in it.

Content filtering can be done either in server-side Web applications or by an intermediate component, such as a proxy server, between clients and a server-side Web application.

Content filtering can be classified into two types: *blacklist-based filtering* and *whitelist-based filtering*.

For example, [55] is an example of the blacklist-based filtering, which intercepts HTTP requests and response at a proxy server, and then escapes dangerous characters such as < and >. However, the proposed method in [55]

would not work in complicated Ajax applications, because legitimate HTTP requests and responses often include HTML tags.

AntiSamy [78] is an instance of the whitelist-based content filter, which requires registering the safe content patterns in advance, and removes the content that does not match the pattern. The whitelist-based filtering is more restrictive but safer against zero-day attacks, although it incurs higher costs for maintaining the legitimate whitelist patterns.

Figure 3.1 shows an example of XSS and its sanitization in a server-side Web application written in Java Server Page (JSP). In this example, the `filterHTML` function escapes a string to make it safe to be inserted in the HTML body. For example, by transforming special characters such as `<` and `>` into `<` and `>`, they will be rendered as string literals rather than as parts of HTML tags. Also, the `filterAttr` function removes a string if it starts with "javascript:", because when such attribute is evaluated as a URI, the string that follows "javascript:" will be executed as a JavaScript code.

Note that the this is a very simple example. Comprehensive countermeasures have to deal with various obfuscation techniques [45] as well as variants of attacks that use other content types such as style sheets and JAR files as described in Section 2.4.3.

Note that content filtering on a server or a proxy cannot prevent the DOM-based XSS, because malicious content (such as a URL with injected script in its request parameters) often would not be sent to the honest servers.

Taint Analysis

Content filtering is error prone because it depends on human developers to insert sanitization code in a Web application. Therefore, *taint analysis* is an effective means to make sure that all user input is sanitized before being rendered into an HTTP response. More detail of taint analysis is discussed in Section 3.7.

3.2.2 Client-Side Technologies

This section describes client-side technologies for preventing XSS, technologies for enabling secure mashups on traditional Web browsers, and emerging effort of redesigning browser security model for next-generation Web browsers.

Client-Side Mechanisms for XSS Prevention

There are many client-side prevention techniques to prevent XSS attacks.

Hallaraker and Vigna [48] enhanced a Web browser to monitor the application API calls to detect suspicious behaviors through anomaly detection. The proposed mechanism monitors the coarse-grained behaviors such as network access and specific browser API calls such as `window.open()` and `window.alert()`. Since it does not track the information flow through script execution, it may fail to detect sophisticated attacks.

Kirda et al., [62] proposed a client-side application-level firewall to detect and prevent XSS attacks, in particular those that send cookies to remote attackers. The browser's protection policy is provided by the user. The proposed approach controls only the network connections and not other browser

behaviors.

Vogt et al, [91] proposed a hybrid approach that combines dynamic and static data flow analysis to prevent XSS attacks on the browsers. The proposed method detects XSS attacks that try to send sensitive information to the attacker. Their method dynamically propagates taint bits during code execution, and associates the taint bits with the DOM nodes. It also uses flow sensitive intra-procedural analysis to detect implicit flows.

BEEP [58] is a countermeasurement for XSS attacks that inserts policies into a mashup server, and enforces the policy at the security-extended browser. Whitelists and blacklists are used to prevent untrusted JavaScript from being executed, but no fine-grained access control is supported for the JavaScript code.

Livshits et al., [69] proposed a type of DOM-level access control that they called "a natural refinement of the same-origin policy". The "principal" information is associated with each DOM node as an attribute, or with a JavaScript function as a property. Basically the code's behavior is limited within the scope of the node for the same principal. There is not much explanation about how the policy is enforced or how it resists attacks.

Content-Security Policy (CSP) is a new proposal by Mozilla [5] to prevent XSS attacks. CSP is the most relevant countermeasure for the secure browser architecture presented in Chapter 4. CSP allows an HTML document to be associated with a security policy file, which specifies the whitelists of the domain names from which content such as JavaScript, images, style-sheets, audio, or video can be downloaded and imported into a webpage.

In CSP, an attempt to download content outside of the whitelists, such

as by using `<script src="...">` will be ignored by the browser unless the action is permitted in the policy. The policy is described in an XML file, and transmitted to the Web browser via the `Policy-URI` HTTP response header.

CSP is a powerful solution to minimize the risks of XSS. At the same time, it imposes strong restrictions that prevent existing Web applications from functioning properly. For example, CSP-enabled browsers will ignore all in-line scripts, such as script embedded inside of the `<script>` elements or attribute (e.g., `<a href='javascript:...' >`), as well as those associated with HTML elements as event handler attributes (e.g., `<body onload='...'>`). Some JavaScript functions, such as `eval()` or `setTimeout()` that allow evaluating and executing a string as JavaScript code are also disabled. Although it is technically possible, it would be realistically difficult to convert many existing Web applications to conform to CSP.

Mashup Frameworks for Traditional Browsers

Several researchers are working on mashup security problems in the existing browser architectures with the same-origin policy. These technologies allow isolating contents that are downloaded from different domains to prevent undesirable interference with each other.

Subspace [57] proposes a secure mashup mechanism that isolates each application in an HTML iframe (inline frame) and then allowing only limited communication between the iframes. In general, applications downloaded from different domains cannot communicate, but most browsers relax the same-origin restriction by allowing overwriting of the built-in `document.domain` property with a super-domain of the original domain name. For example,

two applications from `foo1.bar.com` and `foo2.bar.com` can communicate if they both overwrite the value of the built-in `document.domain` property with `bar.com`.

SMash [61] also isolates applications by using iframes to enable secure mashups on existing browsers. It consists of a JavaScript framework that provides three layered communication channels for cross-domain communication between iframes. SMash uses a method called " fragment communication ", which leverages the fact that the built-in `windows.location` property cannot be read but can be overwritten by iframes even when they belong to different domains. SMash also provides security tokens to protect message integrity and guard against impersonations. Since the upper layers of the communication channels are independent of the implementation of the bottom layer, the fragment communication can be replaced with other mechanisms once the browsers support native cross-domain communications. SMash became an industry standard and was published as open source software as OpenAjax Hub 2.0 [16].

Next-Generation Browsers

There is work underway to redesign the browser architecture to explicitly support secure cross-domain communications. For example, W3C is designing the HTML5 specification as the next generation of HTML, in which a new `postMessage` function is proposed to support explicit communications between frames containing content downloaded from different domains [7].

In addition, the XMLHttpRequest Level 2 Specification [14] proposes extending XMLHttpRequest to support explicit cross-domain network com-

munication. The access control specification [1] that is part of [14] is already implemented in Firefox 3.5 [1]. Microsoft Internet Explorer(R) 8 supports content isolation and explicit cross-domain communication [73].

MashupOS [52] and the module tag [29] propose extensions of the browser architecture to enable explicit sandboxing and cross-domain communication.

3.2.3 Contributions of this Thesis

Server-side Taint Analysis

In this thesis, DIFCA (Chapter 5) can be used for the server-side dynamic taint analysis. An advantage of DIFCA over prior approaches is that it can track and control fine-grained information flow while not requiring modification of existing Web applications. Since the monitoring code is inserted into the Web application code as bytecode instrumentation, the technology is independent of language interpreter (i.e., the Java VM in this case). DIFCA can also provide flexible API-level declassification policies that can untaint values produced by sanitization functions.

A Client-Side Mechanism for Preventing XSS

The secure browser model (Chapter 4) is a client-side technology for preventing XSS attacks. As described earlier, most of the prior approaches can monitor and control only coarse-grained browser behaviors, such as network access, specific built-in APIs, or the execution of JavaScript code. The secure browser model enforces fine-grained access control to browser resources such as DOM nodes and the cookie store, and the privilege of the JavaScript

[1] See https://developer.mozilla.org/En/HTTP_Access_Control

code is determined by the fine-grained policies and the origin of the content, which enables more flexible policy enforcement.

3.2.4 A Browser Model for Secure Mashups

Most of the existing proposals for mashup security are based on the idea of isolating each piece of content within its own sandbox, which is done by using iframes in the current browsers or by extending the browser architecture. Although such approaches are simple and direct, there are drawbacks such as the loss of granularity and a less integrated user experience.

Existing work is based on coarse-grained control. This means content from each domain is isolated into its own sandbox but the capability of each piece of content is not much different from what it is now, and it is not possible to specify a different level of priviledges for each piece of content. In addition, most proposals today separate content based on the *domain*, which is identified by the combination of the server's fully-qualified domain name and port in the same way as the same-origin policy. If multiple applications with different trust levels are hosted on the same server, the coarse-grained approach cannot distinguish among them.

For the user experience, the mechanisms in the existing proposals isolate each piece of content in its own rectangular region in a webpage, and thus two services cannot be fully integrated visually as it can be done in the secure browser model of this theis. In addition, the page layout capabilities by using stylesheets are limited in the sandbox approach. For example, many Ajax libraries implement a dialog box UI with a `<div>` element and a stylesheet, but such a dialog box cannot be displayed beyond the boundaries of an

iframe.

The secure browser model (Chapter 4) proposed in this thesis addresses these problem by using fine-grained DOM-level access control, which allows tight integration of multiple services without isolating them in individual sandboxes. In addition, the proposed model can prevent CSRF and local network attack by malicious services that become part of a mashup application, by controlling their capabilities to access the network.

3.3 Threats against Servers via an Honest User's Web Browser

3.3.1 Technologies for Preventing CSRF and its Variant

Probably the most popular countermeasure for CSRF today is to insert a second authorization token (apart from the cookie that was the first authorization token) into an HTTP request in the form of an HTTP request parameter or POST message, to be checked by the server.

The second authorization token has to be securely inserted into the content of a legitimate Web page, such as a hidden field in an HTML form or directly inserted into a URL, and its value has to be unpredictable so that only a legitimate Web page can issue an HTTP request with a valid token. Johns and Winter proposed an automated mechanism for this approach by using content rewriting at a proxy server [59]. Some application server middleware such as ProjectZero implement this mechanism as an optional

security feature [53].

Another common practice for protecting highly sensitive applications, such as on-line banking applications, is to verify the intent of a legitimate user by authenticating the user again (such as requesting to input the password) before making an important transaction.

Jovanovic et al. [60] proposed a mechanism in which a server-side proxy modifies content to insert the authorization tokens and manages sessions by using the inserted tokens.

Note that not only the static URLs in a webpage but also in all HTTP requests dynamically issued by JavaScript with XMLHttpRequest has to be instrumented to include the second authorization token. Care has to be taken to protect the communication channel, such as by using SSL, to prevent attackers from observing the HTML document and stealing the second token. In addition, the tokens for the Web applications with high security requirements should not be shared with other applications that use non-SSL channels.

The simplest countermeasure for CSRF is to check the `Referrer` HTTP request header on the server to verify that the request was issued from a webpage in the same application. However, it is known that the `Referrer` header is often suppressed due to privacy concerns, either by a browser setting or by the network gateways, and cannot be relied upon.

To avoid the header suppression problem, Barth et al. [17] proposed the `Origin` header as a new header in HTTP requests. Similar to the `Referrer` header, the `Origin` indicates the origin of the content that is issuing the HTTP request, but only at the granularity of the protocol, the server name,

and the port number. Since the `Origin` header conveys less sensitive information than `Referrer`, it is expected to be not suppressed by network gateways. The `Origin` header was introduced in Firefox 3.5.

BEAP [71] is a client-side protection mechanism implemented as a browser extension, which infers whether a request reflects the user's intention and whether an authentication token is sensitive, and strips sensitive authentication tokens from any requests that may not be reflecting the user 's intentions. The inferences are based on information about the request (such as how the request was triggered and crafted) and heuristics derived from analyzing real-world Web applications.

JavaScript hijacking (See Section 2.4.6) is essentially a variant of CSRF, and thus can be prevented by the countermeasures for CSRF. Chess et al. [26] introduced various special techniques to mitigate the risk of JavaScript Hijacking, such as wrapping a JSON string in the JavaScript comment syntax so that the client-side code from a different domain cannot read the content by accessing it via the `<script>` tag. Only the client-side JavaScript in the same-domain can read the content by using the built-in XMLHttpRequest API, and then pre-process the content (such as removing the comment syntax) before parsing the JSON string.

3.3.2 Contributions of this Thesis

Most of these approaches mentioned above require some protection mechanism running on the server-side. However, a Web application server often has unfixed vulnerabilities, and so it is important for users to have a client-side mechanism to protect themselves.

The Secure Browser Model (Chapter 4) is a client-side mechanism that can prevent CSRF attacks based on the well-defined policies, which allows users and Web application owners to express their intent explicitly in advance.

3.4 Threats against Local Networks

3.4.1 Prevention of Local Network Attacks

The traditional Web browser architecture cannot prevent Local Network Attacks. The recommended countermeasures for Cross-Site Printing [94] and Drive-by Pharming [86] are to configure network printers or home routers to authenticate users, and to make sure that the passwords are reasonably unpredictable. In addition, users can prevent the Local Network Attacks combined with CSRF by manually removing the associated cookies after accessing printers or routers.

3.4.2 Contributions of this Thesis

The secure browser model (Chapter 4) is a client-side technology that can prevent Local Network Attacks, by controlling the Web browser's access to the network devices in the local network. A user can specify the policies to protect the local network devices. The policy specification should be straightforward in most cases, since it is rare that a legitimate external Web application needs to access devices in a local network.

3.5 Related Work on Secure Browser Architectures

Web browsers are execution platforms for integrating various pieces of content written not only in HTML and JavaScript but also using Flash, Java, PDF, etc. Since plug-in engines are priviledged to access resources outside of a browser's sandbox, vulnerabilities in the plug-in engines or content will put not only the Web applications but also the computer platforms at risk.

There are also several efforts underway to design secure Web browsers, not only for supporting mashup applications but also to address generic security problems. Here are some examples:

The OP Web browser [40] proposes a Web browser architecture in which components such as a browser kernel, JavaScript engine, and plug-in engines are isolated by using an OS-level sandbox. The architecture minimizes the consequences of vulnerabilities even if they exist.

The Google Chrome browser [18] has a similar structure to the OP Web browser [40], but due to concerns with backward compatibility, the plug-ins are less restricted and allowed to access file systems and networks.

Gazelle [93] is a new browser architecture that provide OS-level isolation at a finer granularity based on *princples*, where principles are determined by the same-origin policy (i.e., determined by the protocol, the server, and the port number). Unlike Chrome, the Gazelle browser isolates objects embedded in an HTML page, such as `<object>`, `<embed>`, `<img>`, and some types of `<input>`. These new browser proposals do not address the fine-grained access control problems discussed in Chapter 4.

The design of JavaScript has long been seen as a problem and many improvements, such as namespace separation and private scopes, have been proposed for ECMAScript4 [15]. However, the standardization committee abandoned most of the new functionalities of ES4 because they did not match the reality of the Web, where backward compatibility is essential, and so the next version of JavaScript will be based on the current ECMAScript version 3[32].

There are also some activities to define safer subsets of JavaScript with object-oriented encapsulation capabilities, such as Caja [74] and ADSafe [2]. JavaScript code that is statically verified to comform to such subset can be executed safely on an existing Web browser environment.

The improved versions of JavaScript do not address browser security issues, although they will advance the secure browser model with their language support for encapsulation and namespace separation.

Tateishi and Tabuchi model the information flow in a browser for confidentiality point of view [88]. Seki [82] proposed DOM-based access control, but did not address the confused-deputy problem discussed in Section 4.3.

3.6 Data Leakage Prevention

There are two major approaches to detect sensitive data flows. The first approach is to use content analysis techniques to identify sensitive data in large amounts of unclassified data. The second approach is to track the flow of data while the data is being processed in a runtime environment, to track the locations of the sensitive data in a program and how it is propated.

This section mainly discusses data leakage prevention technologies based

on content analysis, which corresponds to the proposed technology WebDLP (Chapter 6). Language-based information flow techniques, that are used to protect against the leakage of sensitive data during program execution are discussed in Section 3.7, and are more relevant to DIFCA (Chapter 5).

In general, data leakage prevention solutions are classified into three types: network DLP, host-based DLP, and DLP for data storage.

Network DLP detects and prevents flows of sensitive data at the level of the network protocols. Host-based DLP, often referred to as end-point DLP involves technologies to monitor and control sensitive data at client and server machines. DLP for data-at-rest is a type of end-point technology, but focuses more on identifying and protecting sensitive data in persistent storage, such as by crawling the database and file servers and taking inventory of the sensitive data.

Many of the DLP solutions and products of today focus on network DLP, such as [9], [10], [27], and [87]. Some of them also provide analysis of the application-level protocols, such as filtering the Web on instance messaging by IronPort [27] anord e-mail filtering by PacketSure [87]. Typical DLP solutions as well as content analysis techniques are introduced in [75]. However, due to the nature of security appliances, the detailed mechanisms of these products are not published.

For the HTTP protocol analysis, the most relevant work was presented in [25], which analyzed HTTP requests and responses to estimate the amount of actual information flow.

3.6.1 Contributions of this Thesis

WebDLP (Chapter 6) was inspired by [25] to detect data flows by using protocol analysis, but provides addtional value.

First, WebDLP detects data flows not only between a client and a server but also between different server domains. This analysis addresses the concerns of information leakage in mashup applications. Second, WebDLP not only detects data flow but also classifies the data in transit. This is done by comparing the content of any outgoing data with the classification rules along with a set of pre-classified data. The outgoing data is classified based on its similarities to the known data set, and so WebDLP can detect data leakage that occurs when reusing content, which can be overlooked if simpler pattern matching or exact binary matching algorithms are used.

3.7 Language-Based Information Flow Control

This thesis proposes several technologies to mitigate the data leakage risks of Web applications by using information-flow control technologies. This section provides a brief overview of the prior work on information-flow control, especially for, fine-grained language-based information-flow control.

A number of researchers [24][22][65][30] have studied policy models to guarantee secure information flow. There are also technologies to implement such models. For example, the Multi-Level Security (MLS) system implements the Bell-LaPadula [22][65] model, providing strong isolation from the hardware and networks.

When the information flow control is implemented at the granularity of a system or a process, it is inevitable that securely designed information flow policies cause label-creeping problems [31], since the classification of information becomes more strict thorough propagation, making it increasingly difficult for any information to be shared effectively. Language-based information flow control for fine-grained information flow control is receiving attention. This section briefly reviews prior research work in that area. The language-based information flow control can be classified into static approaches and dynamic approaches. A more thorough survey of static approachs is found in [84].

3.7.1 Static Approaches

Kobayashi and Shirane [64] defined a small subset of Java bytecode and statically analyze the information flow within it. Barthe et al. [19][21][20] proved the noninterference properties for a subset of Java bytecode and proved that programs written in a security-enhanced language can be compiled into bytecode without weakening the security properties. No implementation was reported. Genaim and Spoto [37] studied the information flow analysis of a more complete set of Java bytecode, and implemented their proposed method. Yu and Isam proposed Typed Assembly Language for Confidentiality (TALC) [102] for information flow analysis and proved its noninterference properties.

Jif [76][103][104] is an extension of the Java language that allows defining security labels as types of program constructs. Programs written in Jif are compiled into ordinary Java bytecode, and thus have no dependencies on the run-time environment. However, existing applications need to be converted

into Jif programs. Recently, Boniface et al. implemented an e-mail application in Jif [50] and to evaluate its usefulness in other realistic applications. Li and Zhancewic [67] addressed the information flow problem in Web applications with a security-enhanced scripting language. Static approaches for the declassification problem are found in [77] [66].

3.7.2 Dynamic Approaches

A dynamic approach is potentially more precise than the static approach since t can exploit the detailed conditions of the running programs. It also allows security policies to be defined dynamically.

Beres and Dalton [23] modified an operating system to track information flow in the execution of machine language. A similar approach is applicable to Java, but it requires modification of the Java VM, and the resulting implementation is dependent on the JVM.

Erlingsson et al. [33] proposed inserting Inline Reference Monitors (IRMs) into Java bytecode to implement access control that is equivalent to the Java2 security architecture [38]. Haldar, Chandra, and Franz [46] [47] [35] proposed information flow control for Java using a bytecode rewriting technique. Their initial focus included "taint" propagation for values input into web applications, but can be easily extended to support richer labeling systems. According to Franz [35], they support label propagation at the granularity of Java objects and fields. The dynamic mechanism tracks information propagation at the time of method invocations and field access. They also employ static analysis for detecting implicit flows. Their reports do not say if their approach supports information propagation through all of the JVM instruc-

tions that involve operand stacks and local variables.

3.7.3 Contributions of this Thesis

DIFCA (Chapter 5) was influenced by [46] [47] [35], but the major difference is in the granularity of the label propagation. DIFCA supports information propagation through most of the JVM instructions, including arithmetic operations, array elements, multi-threading, and exceptions, and a policy-based declassification mechanism. DIFCA also integrates an intercepter for database queries, to effectively label database query results and to control input and output to the database through the JDBC.

The problems of implicit flows in a dynamic approach are addressed in [43][85] based on a combination of dynamic and static analysis. Shroff [85] also addresses the problem by analyzing the dependencies of data through monitoring multiple executions of the program. Since our work aims at detecting undesirable information flows without modifying the application source code, it is not the focus of DIFCA to detect all of the implicit flows. However, the author believe that the techniques presented in this thesis can be extended to collect information about the data dependencies and to detect indirect implicit flows.

The secure browser model (Chapter 4) is designed based on the assumptions that the browser supports dynamic information flow tracking capabilities during JavaScript execution and in DOM operations. However, the proposed model is independent of any implementation.

Chapter 4

Information-flow-based Browser Security Model

4.1 Introduction

Ajax offers new modes of Web application construction involving asynchronous data exchanges between the clients (i.e., browsers) and servers, and using JavaScript to update the GUI via the DOM (Document Object Model) and style sheets. Ajax supports interactive user interfaces without reloading the webpages, and this makes possible desktop-application-like user experiences for Web applications. For example, word processors and spreadsheets have been the major native desktop applications on PCs, but they can now be implemented as Ajax applications [6].

Mashup is an application programming model that combines multiple content sources into a single user experience. A typical mashup application might integrate a third-party map service and a list of stores to interchangeably search for the shops near some location on the map, or to find the

locations of the shops on the map.

The de facto security policy of current browsers is called the *same-origin policy* [83]. The same-origin policy assumes that contents (e.g., HTML documents) downloaded from the same server can trust each other, and therefore limits the communication between contents from different origins. The origin of the content is determined by the protocol, port number, and the server name of the content URLs.

Each set of *same-origin content* is confined within a sandbox, which is a browser window or a frame. Each sandbox holds a DOM (Document Object Model) tree representing an HTML document. The same-origin policy prohibits access between DOMs or JavaScript objects that belong to different origins. External script files imported into the HTML document by the `<script src='...' / >` elements are regarded as part of the HTML document, and thus run within the same sandbox as the main document.

The XMLHttpRequest (XHR) is a de facto standard API that is implemented in most Web browsers. XHR allows a client-side script to issue an arbitrary HTTP request to a remote server. The same-origin policy is also applied to XHR, which means an HTTP request can only be issued to a server that belong to the same-origin as the client-side content. However, the emergence of Web 2.0 technologies such as Ajax and Mashup has revealed design flaws of the same-origin policy.

First, the same-origin policy assumes that the content on the same server is mutually trusted, which does not hold for content generated by many users or mashup applications. It is quite possible that the content on the same server includes malicious scripts from some users or third parties. In

fact, Cross-Site Scripting attacks have been the most serious security threat to Web applications for some years [41].

Second, the same-origin sandboxes still need mechanisms to allow communication between content from the different origins. A typical example that bypasses the restriction of XHR is to use the `src` attribute of an <`img`> element to send arbitrary information to an arbitrary remote server, by passing the information as part of the URL request parameters. This technique is widely used in XSS attacks to steal session cookies and send them to remote attackers. If an attacker uses the `src` attribute of a <`script`> element, he can easily implement bi-directional communication between the browser and a remote server, taking advantage of the fact that the returned script content will be executed on the client-side [54]. Since the network accesses via URL references in the HTML attributes are not under the control of the same-origin policy, an attacker can communicate with arbitrary servers without using XHR. Similar loopholes exist in communications between windows and frames on a browser, which are deliberately used by Ajax frameworks [57][61] to enable cross-domain communications.

This chapter proposes a new secure browser model that mitigates the design flaws of the same-origin policy and mitigates the threats to Web applications. In our proposed browser model, all data, i.e., contents received from servers, are associated with the *security label* which identifies the origin of the data. The labels represents the security domain of the URL from which the content is originated. In addition, the user and each content provider can define the *access control policy* to control how content (i.e., script) can access resources, such as the browser's internal DOM tree and the document cook-

ies. The access control policy can also control execution of script as well as network access by JavaScript (e.g., XMLHttpRequest) and HTML elements and attributes (e.g., by using an `<img>` element). The access control policy is defined based on the origin of the content. In order to enforce the access control policy properly in JavaScript, which has dynamic nature, the proposed access control model is built on top of information-flow-based access control (IBAC) [80]. In IBAC, the privilege of the content is judged based on the origin of the data used in each operation. The origin of the data is tracked through script execution. IBAC has advantages over the Stack-based access control (SBAC), a code-origin based access control mechanism which is widely used in Java. Since IBAC assesses the privilege of the content not only based on stack inspection but also the origin of method parameters, our proposed model can enforce access control policies based on the originator of an action even in dynamic and self-mutating client-side Web applications.

The rest of this chapter is organized as follows: Section 4.2 describes some attack scenarios. Section 4.3 offers some observations on the access control models. Section 4.4 describes the detailed design of a secure browser model and its operational semantics. Section 4.5 shows the correctness of the proposed model. Section 4.6 presents an example scenario and evaluates the safety of the model with sample attacks. Section 4.7 makes observation on some design decisions.

4.2 Motivating Scenarios: Cross-Site Scripting

This section describes an attack scenario that is enabled by the shortcomings of the current browser security model, and then gives overview of how the proposed browser model prevents the attacks.

XSS is a type of attack in which an attacker injects malicious script into an innocent web content, allowing the script to execute in a victim's Web browser. XSS can be categorized into three types: *persistent, reflected*, and *DOM-based*.

Persistent XSS. Typically script is injected into a Web message board or blog comments. The injected script is stored persistently in the database, and executed when a victim browses the content.

Reflected XSS. The reflected XSS uses a vulnerability of a server that 'reflects off' unsanitized user input. An attacker may inject a malicious script into the request parameter of a URL, and embed the URL in spam mail, to trick the victim into clicking on the link to trigger the attack.

DOM-based XSS. The DOM-based XSS is possible when the client-side application is designed to read a URL string from `document.location`, `document.URL` or `document.referrer`, and inject part of it into the DOM [63].

Once a malicious script is triggered, it is executed within the *same-origin* sandbox of the innocent content, and therefore allows an attacker to steal

document cookies or user passwords, or compromise data in a Web application for the purpose of phishing [89]. Likewise, Mashup applications have the same vulnerabilities as XSS, because third-party content is often integrated and executed in the same-origin context as trusted content.

A common practice to prevent XSS is to filter out scripts on the server side. However, in case of DOM-based XSS, server-side filtering is not possible because malicious URLs are handled only on the browser side. Even when the server side filtering is logically possible, it is difficult to correctly filter data because new ways of bypassing filters are constantly invented [45][12]. It is reported that 73% of Web applications suffer from XSS vulnerabilities [41]. Therefore, it is important to provdide a mechanism to protect users from vulnerable Web applications.

4.2.1 DOM-Based XSS and Our Approach

This section describes the details of how a DOM-based XSS works, and how our proposed browser model prevents the attack.

1. The user receives a malicious URL, such as `http://foo.com/bar?name=john#<script>...` `...</script>` via a channel outside of the browser, such as in spam e-mail.

2. When the URL is passed to the browser, e.g., by the user double clicking on the link in the spam mail, the browser sends the HTTP request `GET /bar?name=john` to the server foo.com. The fragment identifier after the `#` sign will not be sent to the server.

3. The server foo.com returns a HTML document which includes vulnerable JavaScript in 4.1.

4. The URL of the document, including the fragment identifier, will be assigned to the browser's built in variables such as `document.location`.

5. The returned HTML document is parsed and rendered by the browser, and the embedded JavaScript code is executed. The vulnerable JavaScript code in Fig. 4.1 tries to retrieve the `name` parameter in the URL, and then print it on the page. However, since the value of the variable `s` will include the entire string after `name=`, not only "Welcome john" but also the following script will be inserted into the document. As a result, the script will be executed.

```
<html><body> ..
 <script>
 var pos=window.location.indexOf("name=")+5;
 var s = window.location.substring(pos,
                  document.location.length);
 document.write(" Welcome "  + s);
</script>..
</body></html>
```

Figure 4.1: Example JavaScript vulnerable to DOM-Based XSS

This attack cannot be prevented by the same-origin policy, because the URL string is not explicitly associated with any origins but implicitly trusted. It is implicitly regarded as in the same-origin as the HTML document, even if it comes from an untrustworthy source.

In our proposed browser model, all data, including the URL and any content received from servers, are associated with the *security labels* that

identify the origin of the data. The labels are associated with the URL from which the content originated. Optionally, the *labeling policy*, which is defined by the user and the content providers, can define finer granular labels to parts of the content. In the above example, the URL string that comes from outside of the browser cannot be trusted, and thus is given the label $\top$ which indicates the distrusted status of the data.

The *access control policy*, which is also defined by the user and the content providers, decides how content (i.e., script) can access resources, such as the browser's internal DOM tree and the document cookies. The access control policy can define that content with the label $\top$ does not have permission for script execution.

When the script in Fig. 4.1 is executed, the security label $\top$ of `document.location` will be propagated through the script execution. Therefore, the variable `s` will also have the label $\top$, because it's value will be a substring of `document.location`. As a result, when the variable `s` is written into the DOM by the `document.write` method, the script stored in `s` will not be executed because of the lack of the execute permission for $\top$. Note that the value in `s` is associated with a set of security labels that represent all of the origins on which the value of `s` depends. For example, let's say that the script of Fig. 4.1 is part of an HTML document from `a.com` and associated with a security label `A`. The security label of `s` will be the composition of the label `A` and $\top$, because `s` consists of substrings from both origins. All the security labels associated with `s` need to have execute permission in order to execute the string of `s`.

In addition, in the proposed model, a content provider may associate a finer grained security label with parts of the document. For example, a

provider may associate a different security label with the `<div>` elements which contain user generated messages, and give fewer permissions to that content, in order to mitigate risks of persistent XSS.

4.3 Access Control Models

In order to enable the functionalities depicted in the motivating scenarios, our browser model control access based on the origin of the content (e.g., script).

Stack-based access control (SBAC), or stack-inspection, is an access control mechanism based on the origin of the code. In SBAC, when a privileged function is called, the security manager checks the origin of the functions on the call stack, and in principle, the action is granted only when all of the functions on the call stack have the required privileges. By checking the call stack, the mechanism prevents the confused-deputy attack by malicious code that tries to call privileged functions via trusted libraries. SBAC was introduced in Java2 [38] to control the behavior of untrusted mobile code.

Our browser model uses stack inspection in order to support access control based on the code origin. Stack inspection is necessary to prevent unauthorized code from accessing the sensitive features via trusted JavaScript library code, such as JavaScript widget libraries.

Some implementations of JavaScript have already adopted SBAC in order to enforce the same-origin policy. In particular, Mozilla supports signed JavaScript to grant extra permissions for trusted scripts. However, unlike Java which has namespace separation and private scopes to enable encapsulation, SBAC is not sufficient for JavaScript, which lacks the notion of

the private scope and allows overriding the existing objects and functions. Therefore, an attacker may override objects and their properties, and then compromise the behavior of the trusted code. To prevent such attacks, the Mozilla runtime grants additional permissions to signed JavaScript code only when all of the scripts within a webpage are signed and have those permissions. If scripts are signed by different entities, only the intersection of the permission sets associated with all of the signers is granted [11]. This means that when a webpage includes a piece of unsigned script, no additional permissions are granted to any of the signed JavaScript within the page.

One problem with SBAC is that it cannot prevent the types of attacks that indirectly control the behavior of the code via function parameters [80]. In addition, the dynamic nature of JavaScript makes it more difficult for SBAC to control the privileges of the code. The built-in `eval()` function allows executing an arbitrary string as a piece of JavaScript code, and thus the originator of the string has full control over the program behavior. Dynamic DOM update, e.g., by `document.write()` or by the DOM API, will trigger JavaScript execution when the inserted nodes include scripts, and thus results in the same effect as the `eval()` function. The event-driven programming model of JavaScript also makes it difficult to identify the action initiator by examining the call stack, because the initiator of an action cannot be detected from the call stack when an event handler is invoked.

We propose a novel browser security model based on the Information-flow-Based Access Control (IBAC) [80] to identify the origin of the objects (including data and functions) involved in privileged operations.

IBAC is an access control model which makes access decisions not only

based on the origin of the code on the call stack but also based on the origin of the parameters. IBAC prevents the attacks using data that originated from distrusted code. By applying IBAC to JavaScript, we can overcome the problem of object overriding, and can determine the action initiators even when the objects are overridden.

Fig. 4.2 shows a scenario where IBAC in the browser is effective. This example shows an HTML document `A`, which includes (or mash-up) an external map service `E` with a `<script src='..'>` tag. The node `W` denotes the browser window object. `E` has write permission to the DOM node only under A_3 which is a `<div>` element for rendering the map image. Suppose `E` has no execute permission. When `E` writes malicious script under the node A_3 along with the map data, the script will be stored in the DOM tree as text data, but not executed due to the access control policy. However, if some script in `A` somehow copies the data in A_3 and pastes it in A_5, the script is executed if we only employ the stack inspection. The information-flow based access control can detect the origin of the data in A_3 and prevents script from executing even in such a case.

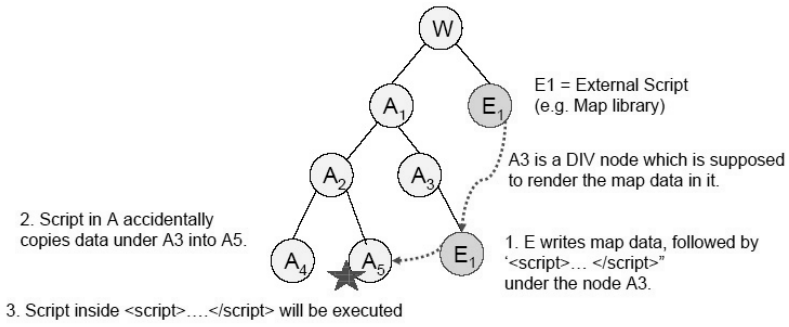


Figure 4.2: Example of Indirect Exection

4.4 Secure Browser Model

This section defines our secure browser model in more detail.

4.4.1 Simplified Browser Model

Table 4.1 shows a simplified model of the Web browser environment and its elements, as well as its JavaScript functionalities.

Table 4.1: Simplified Browser Model

$(World)$	Σ	::=	$\{[u_l = D, \mathcal{P}_D]*\}$
$(Browser)$	B	::=	$\{[w]*, PO, \mathcal{P}_u, \mathcal{K}, \mathcal{H}, \mathcal{L}, \mathcal{P}\}$
$(Window)$	w	::=	$\{D_h, [D]*\}$
$(CookieStore)$	$\mathcal{K}$	::=	$\{[u_l = k]*\}$
$(VariableStore)$	$\mathcal{H}$	::=	$\{[x = D]*\}$
(URL)	u	::=	$u_l \mid u_j$
$(StandardURL)$	u_l	::=	d/p
$(JavaScriptURL)$	u_j	::=	S
$(Domain)$	d	::=	$string$
$(Path)$	p	::=	$string$
$(XPath)$	xp	::=	$string$
$(Data)$	D	::=	$D_h \mid S \mid \ldots$
$(HTMLDocument)$	D_h	::=	$(n*)$
$(DOMNode)$	n	::=	$script \mid img \mid div \mid \ldots$
$(ScriptNode)$	script	::=	<`script`> S <script>
		|	<`script` src=u_l>
$(ImgNode)$	img	::=	<`img src=`u>
$(DivNode)$	div	::=	<`div onevent=`S > $n*$ </`div`>
$(ScriptStatement)$	S	::=	nop $\mid E \mid x = E \mid S; S$
$(Expression)$	E	::=	true | false | string | null
		|	$E \oplus E \mid x \mid$ function $S \mid I$
$(SensitiveInst.)$	I	::=	$readDOM(E_{xp})$
		|	$writeDOM(E_{xp}, E_s)$
		|	$readCki() \mid writeCki(E)$
		|	$f(E*)$
		|	$eval(E) \mid xhr(E_{u_l}, E_s)$
$(PrimitiveOp.)$	PO	::=	$load(u_l) \mid send(u_l, d, k)$
		|	$addChild(n_p, n_c)$

The set Σ is the world of the Web, and represents mappings between URLs u_l and data D, where each data is associated with policy set $\mathcal{P}_D$.

B is a browser instance which consists of windows (i.e., windows or frames) w, each of which contains a main document D_h and multiple sub-

contents D. A sub-content is an in-line image or external script file. The browser also contains 1) a set of primitive operations PO, 2) a cookie store $\mathcal{K}$ for the mapping from a URL u_l to cookie k, 3) a variable store $\mathcal{H}$ for the browser's object store, 4) a label store $\mathcal{L}$ for associating the runtime security labels to objects, such as JavaScript objects and DOM nodes in a given browser state ($\mathcal{L}[x]$ is the security labels for the object x), and 5) a policy store $\mathcal{P}$ that stores the policies activated in the browser, i.e., the browser user's preference and the policies associated with the content. (The labels and the policies are described in the next section)

The URL u may be a standard URL u_l that represents a network location (i.e., u_l is d/p where d is the domain and p is the path) or a virtual `javascript:` URL u_j that includes JavaScript.

Data D is the content downloaded from some URL. An HTML Document D_h includes DOM nodes $n*$. For the sake of simplicity, we consider only JavaScript and passive in-line images as the sub-content that can be imported into the document. The DOM node n may be a `<script>`, `<img>` or other passive elements (such as `<div>`) for formatting purposes. A `<script>` element executes either embedded JavaScript code or an external JavaScript file. An `<img>` element downloads an image from the URL specified in the `src` attribute. Passive elements also may be associated with event handlers. For simplicity, we consider `<div>` to be a passive element and `onevent` is its event handler attribute that has JavaScript code as a value.

Script statements include nop, expression, assignment, and sequences of script statements. Note that conditional branches and loops are intentionally excluded from the model for the sake of simplicity, because we consider only

direct information flows. We do not consider implicit flows [30] in our model, since we assume that content from two origins do not conspire to attack each other, and implicit flows can do very little to compromise the code integrity under this assumption. (E.g., when A does not conspire with E to attack A itself, we think that it is quite difficult for E to control A's behavior by implicit information flow.)

Script expressions include true, false, string, null, binary operations, variable reference, function declaration, and sensitive instruction I. (Note that a function is a first-class object in JavaScript and thus a function declaration is treated as an expression)

I is a set of sensitive operations for which access control decisions are made, including read/write access to DOM nodes and cookies, function calls, execution of arbitrary code by `eval`, and network access with XHR.

PO is a set of a browser's primitive operations: loading a webpage (*load*), sending HTTP requests (*send*), and inserting DOM nodes into the browser window's DOM tree (*addChild*).

4.4.2 Access Control and Labeling Policy

Table 4.2: Browser Policies and Labels

$(PrimitiveLabel)$	l	$::=$	$string_{u_l}$
$(Label)$	L	$::=$	$\{l_1, \ldots\}$
$(LabelPolicy)$	$\mathcal{P}^L$	$::=$	$\{\iota_1, \iota_2, \ldots\}, \iota = (l = u_l \# xp)$
$(ACPolicy)$	$\mathcal{P}^A$	$::=$	$\{\rho_1, \rho_2, \ldots\}, \rho = (s, o, \pm a)$
$(Subject)$	s	$::=$	$l \mid \neg l \mid *$
$(Object)$	o	$::=$	$l \mid \neg l \mid *$
$(Action)$	a	$::=$	$r \mid w \mid e \mid c \mid *$
$(PolicyStore)$	$\mathcal{P}$	$::=$	$\{[\mathcal{P}^L, \mathcal{P}^A]*\}$
$(LabelStore)$	$\mathcal{L}$	$::=$	$\{[D \mid n \mid v = L]*\}$

Table 4.2 shows the *labels* and *policies* that are used in the proposed

browser security model. In particular, the definitions of $l, L, \mathcal{P}^L, \mathcal{P}^A, s, o$ and a correspond to the abstract grammar of the labeling and access control policies.

A primitive label l is a shortname (or an alias) of a URL, and identifies the security domain that a content belongs to. Each content provider has its own namespace for the primitive lables; i.e., a primitive label defined by a content provider is effective only for the access control policy that is provided by the same content provider. In the rest of this chapter, a primitive label defined by an entity a is denoted as $string_a$ where $string$ is the name of the label.

Security label L is a set of primitive labels. We use Denning's lattice model for the security labels [30]. That is, given a finite set of primitive security labels $L_{all} = \{l_1, l_2, \ldots l_n\}$, we construct a lattice for the power set of L_{all}. The top of the lattice $\top = L_{all}$ and the bottom $\bot = \{\}$. A binary operator $\sqcup$ denotes the least upper bound (LUB). The operator $\sqsubseteq$ denotes a partial ordering of the lattice [30].

Policy $\mathcal{P}^L_D = \{\iota_1, \iota_2, \ldots\}$ is a label policy which consists of policy elements $\iota = (l, u_l \# xp)$, each of which associates a primitive label to a URL u_l. Optionally an XPath expression can be used to give a different label on some part of the document. $\mathcal{P}^L[u_l \# xp]$ is the labeling policy of a node specified by the XPath xp in a document downloaded from a URL u_l.

Policy $\mathcal{P}^A$ is the access control policy, which consists of the tuples of a subject s, an object o, and a signed action $\pm a$. The subject is the action initiator, and is identified by the security label of the content such as script. The object o is the target of the action, such as a DOM node n or a network

location u_l. In order to specify a subject or an object in an access control policy, one may also use $*$ to specify all the labels in L_all and $\neg l$ to specify all the labels except for l. Action types include read (r), write (w), execute (e), automatic cookie attachement (c), and $*$ denotes all of them. The read action represents the Web browser behavior of reading (or receiving) data, such as reading data from a DOM or receiving data from a Web server. In contrast, the write action represents the behavior of writing or sending data. The execute action represents the execution of JavaScript code. The automatic cookie attachment action represents the behavior of a Web browser that attaches cookies to HTTP requests. More detailed meanings for these actions are defined in Section 4.4.4.

The sign + or - indicates the positive or negative permission for the action, so the rule $(s, o, -a)$ indicate that the action a is not allowed. We write $(s, o, (+a_1, -a_2))$ as a shorthand of $(s, o, +a_1), (s, o, -a_2)$. The absence of a sign implies +.

A content provider may associate the label policy $\mathcal{P}_D^L$ and access control policy $\mathcal{P}_D^A$ with each piece of data D. Each user may define the user poliies $\mathcal{P}_u^L$ and $\mathcal{P}_u^A$ which are associated with his or her browser instance B.

Browser's label store $\mathcal{L}$ stores mapping from runtime data (such as JavaScript objects, variables and DOM nodes) to the security label L. Note that we use two notions of the *label* in this chapter. First is the *container label*, which is defined in the label policy $\mathcal{P}^L$. The container label identifies the *container* of data, such as a DOM node or a network location. Second is the *data label*, which indicates the origin of the data in a given browser state. The data label is stored in the label store $\mathcal{L}$. The initial data label is determined from

the container label which provides the data, and the data label propagates as the script is executed. On the other hand, the container label of a DOM node or a URL does not change even if data with different labels are written into it.

For example, assume that two HTML documents `a.html` and `e.html` are downloaded from `a.com` and `e.com` respectively. The document `a.html` is associated with the content policy $\mathcal{P}_a^L$ and $\mathcal{P}_a^A$ provided by `a.com`, and the document `e.html` is associated with the content policy $\mathcal{P}_e^L$ and $\mathcal{P}_e^A$ from `e.com`. When $\mathcal{P}_a^L = \{(A_a = a.com), (E_a = e.com)\}$ and $\mathcal{P}_e^L = \{(A_e = a.com), (E_e = e.com)\}$, then DOM nodes in `a.html` and `e.html` will be associated with the container label $L = \{A_a, A_e\}$ and $L' = \{E_a, E_e\}$ respectively. (i.e., each DOM node in `a.html` has the primitive label A_a defined by `a.com` and the primitive label A_e defined by `e.com`).

If `e.html` executes script `var v2="eee"` and then `e.html` execute script `var v1=v2+"aaa"` and then the security label of each data is determined as follows:

1. The data label on string literal "eee" is $\mathcal{L}[\text{"}eee\text{"}] = L' = \{E_a, E_e\}$, because it inherits the container label.

2. The data label on variable $v2$ will be $\mathcal{L}[v2] = L'$ after the assignment because the data propagates from the string literal to the variable.

3. The data label on string literal "aaa" is $\mathcal{L}[\text{"}aaa\text{"}] = L = \{A_a, A_e\}$.

4. The data label on the variable $v1$ will become $(L \sqcup L') = \{A_a, A_e, E_a, E_e\}$ and indicates $v1$ is the composition of the data from `a.html` and `e.html`.

The subject (s) in the access control policy is determined from the composite label of the content involved in the operation, i.e., the composition of the labels of the functions on the call stack as well as the function parameters. The object (o) in the access control policy is identified by the container label of the target object.

For example, in the previous example, suppose that script in `a.html` tries to call a function f_e in *e.html*, and f_e is trying to send an HTTP request to 'http://a.com/'. In such a case, $L = \{A_a, A_e\}$ is the container label of the URL 'http://a.com/' which is the target (object) of the access. On the other hand, the composite label $(L \sqcup L') = \{A_a, A_e, E_a, E_e\}$ is the *subject* of the access; i.e., both `a.html` and `e.html` are involved in the operation. The access is granted and an HTTP request is issued only when $\{(A_a, A_a, (+r, +w)), (E_a, A_a, (+r, +w))\}$ is defined in a.html's policy $\mathcal{P}_a^A$, and
$\{(A_e, A_e, (+r, +w)),$
$(E_e, A_e, (+r, +w))\}$ is defined in e.html's policy $\mathcal{P}_e^A$.

The effective policy set that is enforced at run-time is the composition of the user-defined policies and the policies associated with the stakeholder's content. A stakeholder is some content that is the target of the access. For example, when content in `a.html` accesses the DOM node or cookies in `e.html`, and then the access control policy of `e.html` will be observed.

Definition 1 (Effective Policy Set) *The effective policy set is* $(\mathcal{P}_u^A \cup \mathcal{P}_D^A)$*, where* $\mathcal{P}_u^A$ *is the access control policies in the user preferences, and* $\mathcal{P}_D^A$ *is the access control policies associated with the data* D*.*

Not all DOM nodes or URLs are associated with an explicity container label. Intuitively, when no explicit labeling policy is defined for a DOM node,

an explicit container policy defined for the closest ancestor node is applied. Likewise, for a document at URL u_{l1}, an explicit container policy defined for URL u_{l2} will be applied when u_{l2} has the identical server name and the longest path name prefix.

Formally, we introduce an operator $\succeq$ to denote hierarchical relationships between objects and data.

Definition 2 (Object Hierarchy) *Let o_i and o_j be DOM nodes. $o_i \succeq o_j$ if o_i is an ancestor node of o_j. Let D_i and D_j be documents at the URLs d_i/p_i and d_j/p_j respectively, then $D_i \succeq D_j \iff d_i = d_j \wedge prefix(p_i, p_j)$, where $prefix(x, y) = true$ when the path x is the prefix of the path y. The relation $\succeq$ is reflexive, transitive and anti-symmetric.*

Definition 3 (Explicit and Implicit Container Label) *Let $ep(o)$ and $ip(o)$ be the explicit and implicit container label of an object o (i.e., either a data D or a DOM node n in D, i.e., $D.n$) respectively, and $o.u_l\#xp$ be the URL of D followed by the XPath xp of the node $D.n$. The explicit container label of an object o is L', where $\iota = (L', u'_l\#xp') \in \mathcal{P}_D^L$ where $o.u_l\#xp = u'_l\#xp'$. The implicit container label of o, $ip(o)$ is defined only if $ep(o) = \{\}$ as follows: $ip(o_i) = ep(o_j)$ where $ep(o_j) \neq \phi$ and there is no o_p such that $o_j \succ o_p \succ o_i$ where $ep(o_p) \neq \phi$. If neither explicit nor implicit label is defined, the default label $\top$ is assumed.*

4.4.3 Operational Semantics

Fig. 4.3 and Fig. 4.4 show the semantics of the operations in the big-step style [81]. An operational semantics of the browser is defined by a set of

inference rules, each consists of pre-conditions (above the bar) and post-conditions (below the bar). That is, when a browser operation is being executed, the operation is evaluated as defined in the post-condition, when the pre-condition is satisfied.

We write $(S, C, B) \Downarrow (v, C', B')$ to denote that when script S is executed under the initial call stack C and the browser state B, it evaluates to the value v and terminates with the call stack C' and the browser state B'.

We write $B.\mathcal{H}$ to denote the state of the variable store, and $B.\mathcal{L}$ to denote the state of the label store in the browser. For example, we write $B.\mathcal{L}[v]$ to denote the security label of a value v stored in the browser's label store. We write $B' = B.\mathcal{L}[v \mapsto L]$ when, from an initial state B, the label of object v is updated to L. The browser state after this transition is denoted as B'. Likewise, we write $B' = B.\mathcal{H}[x \mapsto v]$ when the value of variable x is updated to v under the state B, and then the state transits to B'.

Rule 4.1 shows the semantics of when a string literal is evaluated. The value is evaluated as the string itself, and the browser's call stack (C) and the state (B) do not change. The pre-condition is omitted in this rule since this operation does not depend on it. The initial label of a literal value is determined from the container label of the content that includes the literal, and is not explicitly stated in the rule because that simply means that the label does not change. We omit the rules for *true*, $false$ and *null* since they are identical to Rule 4.1.

Rule 4.2 shows the semantics of when a variable reference is evaluated. The value v of the variable x is retrieved from the browser's variable store, such as $v = B.\mathcal{H}[x]$. In addition, the security label L of the variable x is

retrieved from the label store B.$\mathcal{L}$, such as $L = B.\mathcal{L}[x]$, and then stored as the label of the value v, such as $B' = B.\mathcal{L}[v \mapsto L]$. The post-condition shows that the browser state after this operation will be B', which definition is given a part of the pre-condition.

Rule 4.3 shows an assignment operation. First, an expression E is evaluated to a value v, and then the label L of the variable v is taken from the label store $B.\mathcal{L}$, and stored as the label of the variable x. The value v is stored in the variable store with the variable name x, i.e., $B''' = B''.\mathcal{H}[x \mapsto v]$. The post condition of the assignment $x = E$ becomes B'''.

Rule 4.4 shows a binary operation. We write $\oplus$ to represent an arbitrary binary operator. The label of the composite value $v = v_i \oplus v_j$ becomes the Least Upper Bound (LUB) of the labels of the original data v_i and v_j, i.e., $L_i \sqcup L_j$.

Rule 4.5 shows a function declaration. In JavaScript, a function is a first-class object. When a function is declared and its body consists of script S, the label of a declared function object f is determined by the data label of the script S in the function body.

Rule 4.6 shows the execution of a sequential script.

Rest of the rules show execution of a sensitive operation I that requires permission for the corresponding actions. When a sensitive operation I is invoked with the actual arguments $E*$, the permission of the invocation is determined from the composition of all the security labels of script in the call stack C and the arguments. The operation is allowed when each of the data labels in L_s has permission to execute the action to the container object o in the access control policy.

Rule 4.7 shows the operational semantics of $readDOM(E_{xp})$, where E_{xp} is an expression that is evaluated to xp, the XPath expression of the target DOM node. $D_h.u_l$ is the URL of the main HTML document and $D_h[xp]$ is the DOM node specified by an XPath xp. Intuitively, the operation is carried out with the following steps:

1. Argument E_{xp} is evaluated to the XPath xp.

2. The composition of the data labels in the call stack ($\mathcal{L}[C]$) and the argument ($\mathcal{L}[xp]$) is determined. This composite label is represented as L_s in the rule and corresponds to the *subject* (or s) in the access control policy.

3. The container label L_o of the target object, which is the DOM node $D_h[xp]$ that can be identified by the URL and XPath ($Dh.u_l\#xp$) is determined from the labeling policy $\mathcal{P}^L$. The label L_o corresponds to the *object* (or o) in the access control policy. The value in the node is represented as v_n.

4. The permission for reading the DOM node is determined from the access control policy $\mathcal{P}^A$. Intuitively, all primitive labels in L_s need to have read permission for L_o, which is the container label that represents the security domain of the target DOM node. The operation is aborted if L_s has insufficient permissions.

5. If L_s has read permission, then the $readDOM$ operation returns the node value v_n. The browser state after the operation becomes B_3, and the label store $\mathcal{L}$ will be updated such that the security label of the value v_n is equal to the security label of the DOM node $D_h[xp]$.

We introduce the predicate $hasPerm(L_s, L_o, a)$ to validate access permission.

Definition 4 (Permission Check) *$hasPerm(L_s, L_o, a)$ checks whether the access being taken is allowed in the stakeholders' policy that is stored in the policy store $\mathcal{P}^A$. The stakeholders are the user and the entities who's resources are to be protected in the operation. In case of the access to DOM or cookies as well as function calls and the script execution by `eval`, the stakeholders are the user and the HTML document D_h. The stakeholders' policies are the user policy $\mathcal{P}^A_u$ and the content policy $\mathcal{P}^A_{D_h}$. In case of the network access, the stakeholders are the user, the HTML document D_h, and the target network server. I.e., $hasPerm(L_s, L_o, a) = true \iff \forall l \in L_s, \forall sh \in SH : \exists p' = (s', o', +a) \in \mathcal{P}^A_{sh} \wedge \forall l \in L_s : \nexists p'' = (s', o', -a) \in \mathcal{P}^A_{sh}$, where $s' = l \wedge o' = L_o$. SH is the list of the stakeholders and $\mathcal{P}^A_{sh}$ is the access control policy of a stakeholder.*

Rule 4.8 shows the operational semantics of $writeDOM(E_{xp}, E_v)$, where E_{xp} is the XPath of the parent node under which the new node will be appended, and E_v is the value to write. When $writeDOM$ is executed, the browser evaluates E_{xp} and E_v to xp and v respectively, and then parses the value v and generates a set of DOM nodes n_v, i.e., $n_v = parse(v)$. The composite security label (L_s) of the call stack and the argument xp is determined from the label store. The security label of the target node is determined from the labeling policy for $D_h.u_l\#xp$ which identifies the node $D_h[xp]$. The operation is granted only when all primitive labels in L_s have write (or w) permission to the security domain represented by L_o, and the node n_v is inserted into the DOM tree by $addChild(parent, child)$ as described in the

following section. The security label of the argument v propagates to the security label of the generated nodes n_v.

Rule 4.9 shows the operational semantics of $readCki$. The permission is determined by checking if the composite label of the script on the call stack (represented by $L_s = \mathcal{L}[C]$) has read (r) permission for the entire document D_h with the URL represented as $D_h.u_l$. When the access is granted, the cookie v_k is retrieved from the browser's cookie store $\mathcal{K}$. The label store is updated such that the security label of v_k will become the same as the label of the cookie in $\mathcal{K}$.

Rule 4.10 shows the operational semantics of $writeCki$. The permission is determined if the composite label of the script on the call stack has write (w) permission for the entire document D_h, that is identified by $D_h.u_l$. When the access is granted, the value v is written into the browser's cookie store $\mathcal{K}$, assciated with the URL $D_h.u_l$. The security label of the stored cookie becomes the label of the value v.

Rule 4.11 shows a function call. The rule specifies the list of arguments as expressions $E*$ and the values of them as $v*$. Then the composite security label L_s is determined from the call stack and the arguments $v*$. If all the primitive labels in L_s has execute (e) permission on the function object f, then the data label of the function f is pushed to the call stack, and the script S in the function body is evaluated under the updated call stack C'. The value returned from execution of the script S in the function body is v.

Rule 4.12 shows the operational semantics of $eval(E)$, which executes an arbitrary string as a script. When the eval operation is invoked, the browser evaluates the argument E into v_e, and then parses the argument into script,

$S = parsescript(v_e)$. Then the composite security label L_s is determined from the call stack and the argument v_e. The composite security label L_o is determined from the label store on the value v_e. The script S is executed only if all the primitive labels in L_s have execute (e) permission on the value v_e. In that case, the call stack is first updated with the security label of S, and then the script S is evaluated. The permission check on the argument v_e prevents unauthorized string from being executed as a result of injection attack, such as in case of the DOM-based attack.

Finally, Rule 4.13 shows the operational semantics of $xhr(E_{u_l}, E_s)$, which sends the data E_s to the URL E_{u_l} and receives a response. The primitive operation *send* represents the browser's action of sending an HTTP request (See 4.4.4). Since xhr provides explicit bi-directional communication, access to a location u_l by xhr requires both the `write` and `read` permissions.

Intuitively, the access permission of xhr is determined as follows:

1. Arguments E_{u_l} and E_s are evaluated to the URL u_l and the value v_s, respectively.

2. The composition of the security labels in the call stack and the arguments is determined. This composite label is represented as L_s in the rule.

3. The security label L_o of the target object, which is the location u_l, is determined from the labeling policy $\mathcal{P}^L$.

4. The permission for issuing an HTTP request to u_l is determined from the access control policy $\mathcal{P}^A$. Intuitively, all primitive security labels in L_s needs to have read and write permissions to L_o, which represents

the security domain of the URL u_l. The operation is aborted if L_s has no sufficient permissions.

5. The permission for automatically sending cookie (c) is determined from the access control policy. In other words, cookie is sent only when all primitive labels in L_s have the cookie permission c for L_o. (see the detailed algorithm below.) When L_s has insufficient permissions, the HTTP request is sent without the cookie.

When an HTTP request is issued, the `cookie` permission for the target URL is determined, and the cookie is associated with the HTTP request only when the subject has the automatic cookie (c) permission. This restriction prevents the Cross-Site Request Forgery (CSRF) attacks, which details will be described in Section 4.6.

Definition 5 (Permission Check for Cookies)
$allowedCookie(L_s, u_l) = \mathcal{K}[u_l] \iff \forall l \in L_s, \exists p' = (s', o', +c) \wedge \forall l \in L_s, / \exists p''(s', o', -c)$, *where* $s' = l \wedge o' = \mathcal{L}[u_l]$. *Otherwise* $allowedCookie(L_s, u_l) = null$.

4.4.4 Browser's Primitive Operations

The browser's primitive operations are defined in Fig. 4.5. The operation $send(url, data, k)$ represents the browser's action of sending an HTTP request and receiving the response data v_r, with optional content policy $\mathcal{P}'_{vr}$ and cookie k' in response (Rule 4.14). The send operation will update the policy store $\mathcal{P}$ and the cookie store $\mathcal{K}$ with $\mathcal{P}'_{vr}$ and k'. The initial security label of the cookie for the URL, $\mathcal{K}[u_l]$, is equal to the label of D_h.

$$\frac{}{(string, C, B) \Downarrow (string, C, B)} \text{(Literals)} \tag{4.1}$$

$$\frac{v = B.\mathcal{H}[x] \qquad L = B.\mathcal{L}[x] \qquad B' = B.\mathcal{L}[v \mapsto L]}{(x, C, B) \Downarrow (v, C, B')} \text{(Variable Reference)} \tag{4.2}$$

$$\frac{(E, C, B) \Downarrow (v, C, B') \qquad L = B.\mathcal{L}[v] \qquad B'' = B'.\mathcal{L}[x \mapsto L] \qquad B''' = B''.\mathcal{H}[x \mapsto v]}{(x = E, C, B) \Downarrow (v, C, B''')} \text{(Assignment)} \tag{4.3}$$

$$\frac{\begin{array}{c}(E_i, C, B) \Downarrow (v_i, C, B') \qquad (E_j, C, B') \Downarrow (v_j, C, B'') \\ L_i = B'.\mathcal{L}[v_i] \qquad L_j = B''.\mathcal{L}[v_j] \qquad v = v_i \oplus v_j \qquad B''' = B.''\mathcal{L}[v \mapsto L_i \sqcup L_j]\end{array}}{(E_i \oplus E_j, C, B) \Downarrow (v, C, B''')} \text{(Binary Operation)} \tag{4.4}$$

$$\frac{B' = B.\mathcal{L}[f \mapsto B.\mathcal{L}[S]]}{(function\ S, C, B) \Downarrow (f, C, B')} \text{(Function Declaration)} \tag{4.5}$$

$$\frac{(S_1, C, B) \Downarrow (v_1, C, B') \qquad (S_2, C, B') \Downarrow (v_2, C, B'')}{(S_1 : S_2, C, B) \Downarrow (v_2, C, B'')} \text{(Script Sequence)} \tag{4.6}$$

Figure 4.3: Operational Semantics 1

The primitive operation $load(u_l)$ (Rule (4.15)) represents the browser's behavior of navigating to a URL by following a hyperlink or by the user's action of entering a URL in the browser's address bar. The load action may be initiated by other desktop applications that support hyperlinks. When loading an HTML document at the URL u_l into a browser window w, the browser issues an HTTP request to the URL. When a URL string is provided from an external entity (e.g., a desktop mail application), the URL cannot be trusted and thus the default label $\top$ is associated with it. Otherwise, when a page D'_h is loaded as a result of hyperlink in another webpage D_h, the label on the URL string in the content of D_h will be inherited, and recorded as the security label of the document URL $D'_h.u_l$. The label on the document URL prevents DOM-based XSS, because the permission associated

(ReadDom)

$$\frac{\begin{array}{c}(E_{xp}, C, B_1) \Downarrow (xp, C, B_2) \qquad L_s = B_2.\mathcal{L}[C] \sqcup B_2.\mathcal{L}[xp] \\ L_o = B_2.\mathcal{P}^L[D_h.u_l \# xp] \qquad (D_h[xp], C, B_2) \Downarrow (v_n, C, B_2) \qquad B_3 = B_2.\mathcal{L}[v_n \mapsto \mathcal{L}[D_h[xp]]]\end{array}}{\begin{array}{c}(readDOM(E_{xp}), C, B_1) \Downarrow (v, C, B') \\ where \qquad v = v_n, B' = B_3 \qquad if\ hasPerm(L_s, L_o, r), \\ v = null, B' = B_2 \qquad otherwise\end{array}} \tag{4.7}$$

(WriteDom)

$$\frac{\begin{array}{c}(E_{xp}, C, B_1) \Downarrow (xp, C, B_2) \\ (E_v, C, B_2) \Downarrow (v, C, B_3) \qquad L_s = B_3.\mathcal{L}[C] \sqcup B_3.\mathcal{L}[xp] \qquad L_o = B_3.\mathcal{P}^L[D_h.u_l \# xp] \\ n_v = parse(v) \qquad B_4 = B_3.\mathcal{L}[n_v \mapsto \mathcal{L}[v]] B_5 = B_4 \cup addChild(D_h[xp], n_v)\end{array}}{\begin{array}{c}(writeDOM(E_{xp}, E_v), C, B_1) \Downarrow (null, C, B') \\ where \qquad B' = B_5 \qquad if\ hasPerm(L_s, L_o, w), \\ B' = B_4 \qquad otherwise\end{array}} \tag{4.8}$$

(readCki)

$$\frac{v_k = \mathcal{K}[D_h.u_l] \qquad L_s = B.\mathcal{L}[C] \qquad L_o = B.\mathcal{P}^{\mathcal{L}}[D_h.u_l] \qquad B_1 = B.\mathcal{L}[v_k \mapsto B.\mathcal{L}[B.\mathcal{K}[D_h.u_l]]]}{\begin{array}{c}(readCki(), C, B) \Downarrow (v, C, B') \\ where \qquad v = v_k, B = B_1 \qquad if\ hasPerm(L_s, L_o, r) \\ v = null, B' = B \qquad otherwise\end{array}} \tag{4.9}$$

(writeCki)

$$\frac{\begin{array}{c}(E_v, C, B) \Downarrow (v, C, B_1) \\ L_o = B_1.\mathcal{P}^L[D_h.u_l] \qquad L_s = B_1.\mathcal{L}[C] \qquad B_2 = B_1.\mathcal{K}[D_h.u_l \mapsto v] \qquad B_3 = B_2.\mathcal{L}[\mathcal{K}[D_h.u_l] \mapsto \mathcal{L}[v]]\end{array}}{\begin{array}{c}(writeCki(D_h, E_v), C, B) \Downarrow (null, C, B') \\ where \qquad B' = B_3 \qquad if\ hasPerm(L_s, L_o, w), \\ B' = B_1 \qquad otherwise\end{array}} \tag{4.10}$$

(Function Call)

$$\frac{\begin{array}{c}f = S \qquad (E*, C, B_1) \Downarrow (v*, C, B_2) \\ L_s = B_2.\mathcal{L}[C] \sqcup B_2.\mathcal{L}[v*] \qquad L_o = B_2.\mathcal{L}[f] \qquad C' = C \cup B_2.\mathcal{L}[f] \qquad (S, C', B_2) \Downarrow (v, C', B_3)\end{array}}{\begin{array}{c}(f(E*), C, B) \Downarrow (v', C, B') \\ where \qquad v' = v,\ B' = B_3 \qquad if\ hasPerm(L_s, L_o, e), \\ v' = null,\ B' = B_2 \qquad otherwise\end{array}} \tag{4.11}$$

(Eval)

$$\frac{\begin{array}{c}(E, C, B_1) \Downarrow (v_e, C, B_2) \qquad L_s = B_2.\mathcal{L}[C] \sqcup B_2.\mathcal{L}[v_e] \\ L_o = B_2.\mathcal{L}[v_e] \qquad S = parsescript(v_e) \qquad (S, C, B_2) \Downarrow (v, C', B_3) \qquad C' = C \cup L_s\end{array}}{\begin{array}{c}(eval(E), C, B) \Downarrow (v', C, B') \\ where \qquad v' = v, B' = B_3 \qquad if\ hasPerm(L_s, L_o, e), \\ v' = null, B' = B_2 \qquad otherwise\end{array}} \tag{4.12}$$

(xhr)

$$\frac{\begin{array}{c}(E_{u_l}, C, B_1) \Downarrow (u_l, C, B_2) \qquad (E_s, C, B_2) \Downarrow (v_s, C, B_3) \qquad k = allowedCookie(L_s, l) \\ L_s = B_3.\mathcal{L}[C] \sqcup B_3.\mathcal{L}[u_l] \qquad L_o = B_3.\mathcal{P}^L[u_l] \qquad (send(u_l, (u_l, v_s), k), C, B_3) \Downarrow (v_r, C, B_4)\end{array}}{\begin{array}{c}(xhr(E_l, E_s), C, B_1) \Downarrow (v, C, B') \\ where \qquad v = v_r, B' = B_4 \qquad if\ hasPerm(L_s, L_o, (r, w)), \\ v = null, B' = B_3 \qquad otherwise\end{array}} \tag{4.13}$$

Figure 4.4: Operational Semantics 2

with the originator of a URL string will be inherited by the values derived from the URL. The returned data is parsed into DOM nodes, and each node is associated with the security labels (*putlabel*) as defined in the labeling policy $\mathcal{P}^L$, and inserted into the document tree.

Rule (4.16) shows that the script S in an `onevent` attribute is executed when an event occurs. The call stack is empty when an event handler is invoked, and thus only the execute (e) permission of the attribute value is determined.

Rules between 4.17 and 4.21 show the browser's *addChild* behavior, which inserts a node into the DOM tree. The corresponding permissions are checked when the node insertion results in HTTP requests or script invocation.

When an inserted node includes only passive data (such as text or formatting elements), the DOM node is simply updated (Rule 4.17). When the node inserted by `addChild` includes an `<img>` element with a `src` attribute which includes a location URL u_l, that initiates an HTTP request to the URL u_l (Rule 4.18). The behavior is regarded as a one-way communication (given that the returned image is static) and thus requires the `write` permission for the location u_l. If the inserted `<img>` element has a JavaScript u_j, then it requires the `execute` (e) permission to execute the code (Rule 4.19). When the inserted node includes an embedded script, then the `execute` permission is needed to invoke the script (Rule 4.20). If the inserted `script` node has a URL reference in the `src` attribute, `read`, `write`, and `execute` permissions are required before the browser issues an HTTP request for the URL and execute the script (Rule 4.21).

(HTTP Request/Response)

$$\frac{B_2 = B_1.\mathcal{P} \cup \mathcal{P}_{v_r} \qquad B_3 = B_2.\mathcal{K}[u_l \mapsto k'] \qquad B_4 = B_3.\mathcal{L}[\mathcal{K}[u_] \mapsto \mathcal{P}^L[u_l]] \qquad B_5 = B_4.\mathcal{L}[v_r \mapsto \mathcal{P}^L[u_l]]}{\begin{array}{c}(send(u_l, v_s, k), C, B_1) \Downarrow (v_r, C, B_5),\\ where\ v_r, \mathcal{P}_{v_r}, and\ k'\ are\ data,\ content\ policy,\\ and\ cookies\ received\ in\ the\ HTTP\ response.\end{array}} \tag{4.14}$$

(Page Loading)

$$\frac{\begin{array}{c}k = allowedCookie(\mathcal{L}[u_l], u_l)\\ B_1 = B \cup D_h.u_l = u_l \qquad B_2 = B_1.\mathcal{L}[D_h.u_l \mapsto \mathcal{L}[u_l]] \qquad (send(u_l, u_l, k), C, B_2) \Downarrow (v_r, C, B_3)\\ D_h = parse(v_r) \qquad B_4 = B_3 \cup putlabel(D_h, \mathcal{P}^L)) \qquad (addChild(w, D_h), C, B_4) \Downarrow (null, C, B_5)\end{array}}{(load(u_l, w), C, B) \Downarrow (null, C, B_5)} \tag{4.15}$$

(OnEvent)

$$\frac{n.onevent = S \qquad (S, C, B) \Downarrow (v, C, B')}{\begin{array}{c}(S, C, B) \Downarrow (v, C, B'') \qquad when\ an\ envet\ occurs\ at\ the\ node\ n\\ where \qquad B'' = B' \qquad if\ hasPerm(\mathcal{L}[n], \mathcal{L}[n], e), \qquad B'' = B \qquad otherwise\end{array}} \tag{4.16}$$

(add passive node)

$$\frac{B' = B.D_h \cup n_c}{(addChild(n_p, n_c), C, B) \Downarrow (null, C, B')} \tag{4.17}$$

(add img node)

$$\frac{\begin{array}{c}n_c = img \qquad n_c.src = l \qquad L_o = \mathcal{P}^L[n_c.src] \qquad L_s = \mathcal{L}[C] \sqcup \mathcal{L}[n_c]\\ B_2 = B_1.D_h \cup n_c \qquad k = allowedCookie(L_s, n_c.src) \qquad (send(n_c.src, n_c.src, k), C, B_2) \Downarrow (v, C, B_3)\end{array}}{\begin{array}{c}(addChild(n_p, n_c), C, B_1) \Downarrow (null, C, B') \qquad where \qquad B' = B_3 \qquad if\ hasPerm(L_s, L_o, w),\\ B' = B_2 \qquad otherwise\end{array}} \tag{4.18}$$

(add img node with script url)

$$\frac{\begin{array}{c}n_c = img \qquad n_c.src = j \qquad L_o = \mathcal{P}^L[n_c.src]\\ L_s = \mathcal{L}[C] \sqcup \mathcal{L}[n_c] \qquad S = parsescript(n_c.src) \qquad B_2 = B_1.D_h \cup n_c \qquad (S, C, B_2) \Downarrow (v, C, B_3)\end{array}}{\begin{array}{c}(addChild(n_p, n_c), C, B_1) \Downarrow (null, C, B') \qquad where \qquad B' = B_3 \qquad if\ hasPerm(L_s, L_o, e),\\ B' = B_2 \qquad otherwise\end{array}} \tag{4.19}$$

(add script node with embedded script)

$$\frac{\begin{array}{c}n_c = script \qquad L_o = \mathcal{L}[n_c]\\ L_s = \mathcal{L}[C] \sqcup \mathcal{L}[n_c] \qquad S = parsescript(n_c.text) \qquad B_2 = B_1.D_h \cup n_c \qquad (S, C, B_2) \Downarrow (v, C, B_3)\end{array}}{\begin{array}{c}(addChild(n_p, n_c), C, B_1) \Downarrow (null, C, B')\\ where \qquad B' = B_3 \qquad if\ hasPerm(L_s, L_o, e), \qquad B' = B_2 \qquad otherwise\end{array}} \tag{4.20}$$

(add script node with src attribute)

$$\frac{\begin{array}{c}n_c = script\\ L_o = \mathcal{P}^L[n_c.src] \qquad L_s = \mathcal{L}[C] \sqcup \mathcal{L}[n_c] \qquad B_2 = B_1.D_h \cup n_c \qquad k = allowedCookie(L_s, n_c.src)\\ (send(n_c.src, n_c.src, k), C, B_2) \Downarrow (v_r, C, B_3) \qquad S = parsescript(v_r) \qquad (S, C, B_3) \Downarrow (v, C, B_4)\end{array}}{\begin{array}{c}(addChild(n_p, n_c), C, B_1) \Downarrow (v', C, B') \qquad where \qquad v' = v_r, B' = B_4\\ if(hasPerm(L_s, L_o, (w, r)) \wedge hasPerm(L_o, L_s, (w, r))\\ \wedge hasPerm(L_s, L_o, e)),\\ v' = null, B' = B_2 \qquad otherwise\end{array}} \tag{4.21}$$

Figure 4.5: Primitive Operations

4.5 Enforcement of Access Control

We prove the correctness of the model using two theorems: determination of the content origin and enforcement of access control on sensitive operations.

Theorem 1 (Origin of Static and Dynamic content) *The origin of the static or dynamic content that are involved in an operation is determined from the effective labeling policy $\mathcal{P}^L$ the direct information flows during script execution.*

Definition 6 (Content involved in an operation)

The content that is involved in an operation is the script (or functions) on the call stack and the sensitive parameters of the operation.

Definition 7 (Static Content) *The static content is the HTML elements and the script code in D_h and D.*

Lemma 1 (Origin of Static Content) *From Definition 1, 2 and 3 as well as rules (4.14) and (4.15), all data received as an HTTP response will be associated with the initial data label that represents the data origin, according to the policy in $\mathcal{P}^L$. When no labeling policy is defined, the initial label of the data is $\top$ from Definition 3.*

From rules (4.1) and (4.5), the data label of the static content is the initial label. From rules (4.2), (4.4), and (4.3) the data labels are propagated when direct information flows occur within a function. From rule (4.11), labels are propagated as data propagates through function argument and return values, and the call stack is updated as a function is called. From rules (4.7), (4.8), (4.9) and (4.10), the label of the data is propagated as data is stored or retrieved from the DOM or cookie store, and thus origin of the can be tracked.

Definition 8 (Dynamic Content) *Dynamic content is HTML elements or script code that is executed as a result of the DOM node update or the eval function.*

Proof of Theorem 1 *In addition to Lemma 1, the origin of the eval argument is determined as shown in rule (4.12). From rule (4.8), the origin of the inserted DOM node is determined from the origin of the original values.*

Theorem 2 (Enforcement of AC policy) *All of the sensitive operations, i.e., network access, read or write operations to the DOM and cookies, and the execution of script, are controlled by the effective policy set, based on the origin of the static or dynamic content that involve in the operation.*

Definition 9 (Sensitive Operations) *The sensitive operations are* `xhr`, `readCki`, `writeCki`, `readDOM`, `writeDOM`, `eval`, *any script execution, any network access caused by URL references in HTML, and automatic cookie attachment in HTTP requests. (However, page loading by a user is not a sensitive operation.)*

Lemma 2 (AC Enforcement of Static Content) *From Rule 4.13-4.10, permission check by Definition 4 is performed before carrying out the operations in static content. Rules (4.18),(4.20) and (4.21) show that network access by URL references in the HTML will have permission checks by Definition 4. Rules (4.13), (4.18), and (4.21) show that cookie permission is checked by Definition 5.*

Proof of Theorem 2 *Dynamic content execution by* `eval` *will have a permission check as shown in rule 4.12. The* `writeDOM` *operation (rule 4.8)*

will cause DOM update, that result in on of the `addChild` *rules, i.e., (4.17), (4.18),(4.20), or (4.21). Dynamic content execution as a result of DOM updates are modeled in rules (4.18),(4.19), (4.20) and (4.21), and thus subject to the permission checks of Definitions 4 and 5. From Lemma 2s, Definition 4 and 5, and Theorem 1 all sensitive operations, either static or dynamic, are subject to access control by the effective policy set.*

4.6 Examples

This section shows how the proposed browser security model mitigate risks of distrusted third-party services in a mashup application. In addition, we illustrate that the proposed security model can prevent other common Web application threats such as the Cross-Site Request Forgery (CSRF), and local network attacks.

4.6.1 Mashup Application Example

The example environment in Fig. 4.6 consists of a mashup application `a.com`, which integrates an external service from `e.com`. The user's browser is in the intranet which includes an intranet server `u.com`, which may become a target of local network attacks. Fig. 4.7 (1) shows the content of the mashup application at `a.com/a.html` which includes the external content from `e.com` in two ways: 1) some content from `e.com` is integrated into the web page generated by `a.com`, in the form of server-side mashup, and 2) a script file at `e.com/e.js` (Fig. 4.7 (2)) is imported into `a.html` by a script tag. We assume that `a.com` is trusted by the user for the service it provides,

while e.com may potentially be a malicious third party. At the same time, e.com does not fully trust a.com either. Both a.com and e.com use a trusted JavaScript widget library from g.com.

Now we assume that the environment consists of three sets of policies from stakeholders: 1) the user-defined policy, whose purpose is to protect the user and the intranet applications from attacks, 2) the content policy by a.com, whose purpose is to protect its content from potentially malicious external services (such as e.com), and 3) the content policy by e.com whose purpose is to protect itself from potentially malicious mashup applications (such as a.com). Since g.com is a generic JavaScript library, it does not claim any policies.

Fig. 4.8 shows the set of example policies defined based on the policy grammar of Table 4.2. Note that $\mathcal{P}^L_{D_a}$ and $\mathcal{P}^A_{D_a}$ are the labeling and access control policies defined for the document D_a, while $\mathcal{P}^L_{D_e}$ and $\mathcal{P}^A_{D_e}$ are defined for D_e. (The numeric subscript of the policy element is the index in each set of policies). The user policy 1 defines the label U_u for the intranet applications in u.com. The user policy 2 disallows all Web applications, except for those labeled with U_u, to access the intranet applications on u.com, in order to prevent local network attacks. The user policy permits all interactions between other contents that do not originate from u.com. Policy 3 is the labeling policy defined by a.com. The content from a.com is labeled as A_a (Note that the suffix $_a$ denotes that the label is defined by a.com). However, the content under the particular elements, which are expressed in the XPath //*[@class='e'], are labeled as AE_a. The policy also defines that the content from e.com and g.com are labeled as E_a and G_a respectively. Policy 4 is

the access control policy defined by `a.com`, which gives the content with the label A_a full access permissions to the content with the same label A_a. The policy also allows A_a to execute content with the label E_a and G_a, implying that A_a may invoke services provided by E_a and G_a in the form of function calls. Permission of E_a is limited to the node with the label AE_a. The library G_a is allowed read and write permission for both A_a and AE_a. Policy 5 is the labeling policy defined by `e.com`, which defines `e.com`, `a.com` and `g.com` as E_e, A_e and G_e respectively. Policy 6 is the access control policy defined by `e.com`, which gives E_e full access permission for itself, and A_e to read and write access to E_e. The execute permission allows A_e to call functions in `e.com`.

Now, let's see how permissions are granted for normal operations.

1. `e.com/e.js` defines a function `updateMap` which takes a street address, queries the map data from the server `e.com`, and update a DOM node `<div id='e'>` with the map data. The data label of the `updateMap` function object is $\{E_a, E_e\}$, because it inherits the container label on `e.com/e.js`. Now, suppose `a.com/a.html` calls `updateMap` with a string literal,(Fig.4.7(1) Line 5). The `updateMap` function first 1) retrieve data from e.com with XHR, and then 2) write the data into the DOM node (Fig.4.7(2) Line 2-3). The data label of the address string is $L_1 = \{A_a, A_e\}$ because it is originated from the content `a.html`. The composite data label of the call stack will be $L_2 = \{A_a, A_e, E_a, E_e\}$ because the call stack include content from both `a.html` and `e.js`.

   (a) When `updateMap` calls xhr, the subject of the operation is identified by the set of labels $L_s = L_1 \sqcup L_2 = \{A_a, A_e, E_a, E_e\}$. The

object of the operation is $\{E_a, E_e\}$, the container label on `e.com`. The stakeholder is `e.com`, and thus its policy 6 is respected. Finally, the operation is granted since policy 6 allows both A_e and E_e to have read and write access to E_e. Assuming that returned data does not have explicit policy associated with it, it will have the same label as `e.com/e.js` as implicit label (i.e., the label of the map data will be $\{E_a, E_e\}$).

(b) Likewise, when `updateMap` calls *writeDOM*, the subject of the operation is identified by $L_s = \{A_a, A_e, E_a, E_e\}$. The target of the operation is the DOM node `<div id='e'>`, and identified by its container label $L_o = \{AE_a, A_e\}$. In this case, `a.com/a.html` is the stakeholder of the operation because it owns the DOM tree, and thus its policy 4 is respected. The *writeDOM* operation will be granted because policy 4 allows both A_a and E_a write access to AE_a.

2. The content `a.html` can leverage the widget library `G`, to modify the DOM tree (policy 4-4, 4-5).

3. The content from `u.com` has full access to itself (policy 1-1, 2-1).

Now, assume that `e.js` tries to alter the behavior of `a.html` by overriding some variables (`e1,e2,e3`) used by `a.html`.

1. When `a.html` concatinate `e1` with some string literal and calls *eval* on it, both of the subject label are $L_s = L_o = \{A_a, A_e, E_a, E_e\}$ (Note that the object labe in this case is the data label on the parameter of *eval*, which is the concatination of string). The stakeholder is `a.html` and

thus its policy 4 is respected. However, the script will not be executed since policy 4 does not imply neither $(A_a, E_a, +e)$ nor $(E_a, A_a, +e)$.

2. When `a.html` tries to write `e2` into a DOM node, it consists of two steps.

   (a) First, in *writeDOM* operation, the data label of the subject is $L_s = \{A_a, A_e\}$ and the container label of the target DOM node is $L_o = \{A_a, A_e\}$. The data label of the second parameter (which does not effect the subject label) is $L_v = \{E_a, E_e\}$. The stakeholder of the operation is `a.html`. The write operation will be granted since A_a has write permission for A_a.

   (b) And then, the *addChild* operation (Rule 4.20 in Fig. 4.5) will be invoked when the new node is inserted, because the node being inserted is a script node with embedded script. In this case, the subject label is the composition of the labels on the call stack and the parameter, i.e., $L_s = \{A_a, A_e, E_a, E_e\}$. The target of the operation is the data label on the new node, i.e., $L_o = \{E_a, E_e\}$. Again, `a.html` is the stakeholder, and according to its policy 4, the node will be added but the included script will not be executed due to the lack of permission ; i.e., A_a does not have an execute permission on E_a.

Now, let's see how other attacks are prevented.

1. If the application `a.com/a.html` has a DOM-based XSS vulnerability, the URL string will be assigned the label $\top$, which is associated with

no permissions, and thus the injected script will not be executed, as we have described in Section 4.2.

2. `e.com/e.js` may be a malicious content which tries to attack `a.com/a.html`. But `e.com/e.js` cannot read or write the DOM tree of `a.com/a.html`.

3. `e.com/e.js` may modify or read the node under `<div id='e'>` by using the trusted library from `g.com`. However, `g.com`'s library cannot access other nodes in `a.com/a.html` if it is invoked by `e.com/e.js`, or vise versa, due to the stack inspection.

An attacker in `e.com` or `a.com` may try to circumvent the access control by overriding data or content as follows:

1. (Object overriding)`e.com` may override JavaScript object properties to alter the behavior of `a.html`. In the proposed security model, JavaScript objects are not the target of access control, and thus `e.com` may override them. However, the object properties written by `e.com` will have the label $\{E_a, E_e\}$ associated with it. Therefore, even if script in `a.html` is going to invoke some operation with the overridden object, the *subject* of the operation will be $\{A_a, A_e\} \sqcup \{E_a, E_e\}$; i.e., the sensitive operation that involves the overridden object will require permission of both `a.com` and `e.com`.

2. (Fake labels) `e.com` may try to override the labeling policy to allow its access to `a.com`. E.g., by adding $(A, e.com/)$ into the policy set 5. However, in the proposed model, label names defined by different content policy are distinguished. The labels are aliases for the URLs,

and are translated into URL when it is evaluated in the corresponding access control policy.

3. (Fake policies) `e.com` may try to override the access control policies to allow its access to `a.html`. E.g., by adding $(E_e, A_e, (+r, +w, +c))$ into the policy set 6. However, when `e.com` tries to access a DOM node of `a.html`, `a.com` will be the stakeholder who's policy will be observed. Since `a.com` does not allow $(E_a, A_a, (+r, +w, +c))$, the operation will not be granted.

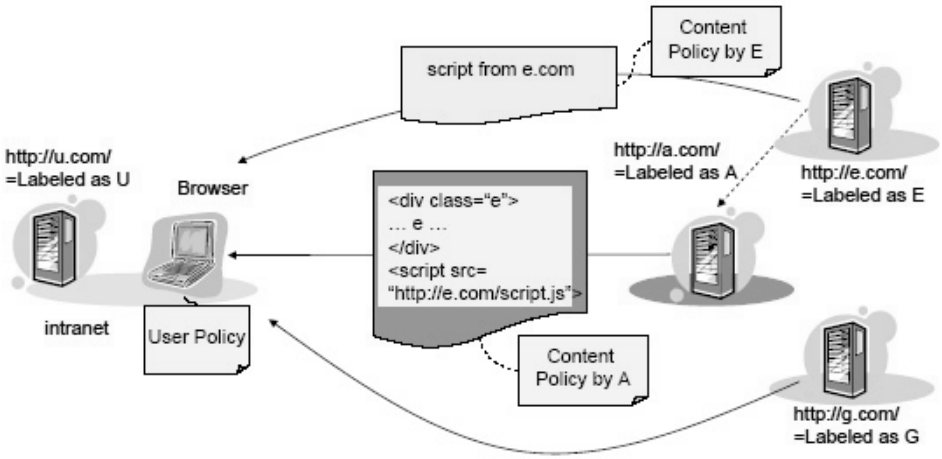


Figure 4.6: Example Environment

4.6.2 Cross-Site Request Forgery (CSRF)

CSRF misuses cookies and HTTP authorization to issue illegal commands by bypassing the request authentication. Many applications track the user's

```
(1) a.com/a.html
1: <script src="http://e.com/e.js">
2: <div class="e"> .. e .. </div>
3: <div class="a"> .. a .. </div>
4: <script>
5: updateMap("1-2-3 Tokyo, Japan");
6: var a1 = "myfunc();" + e1;
7: eval(a1);
8: writeDOM("//*[@class='a']", e2);
</script>

(2) e.com/e.js
1: function updateMap(addr){
2:  var v=xhr("http://e.com/getdata", addr);
3:  writeDOM("//*[@class='e']", v);
4: }
5: var e1="alert('e1');";
6: var e2="<script>alert('e2');</script>";
..
```

Figure 4.7: Example Content

1. $\mathcal{P}_u^L = \{(U_u, u.com)_1\}$
2. $\mathcal{P}_u^A = \{(U_u, U_u, +*)_1, (\neg U_u, \neg U_u, +*)\}$
3. $\mathcal{P}_{Da}^L = \{(A_a, a.com/)_1, (AE_a, a.com/\#// * [@class = "e"])_2,$
   $(E_a, e.com/)_3, (G_a, g.com/)_4\}$
4. $\mathcal{P}_{Da}^A = \{(A_a, A_a, +*)_1, (A_a, AE_a, +w)_2, (E_a, AE_a, +*)_3,$
   $(G_a, AE_a, (+w, +r))_4, (G_a, A_a, (+w, +r))_5, (A_a, E_a, +e)_6,$
   $(A_a, G_a, +e)_7\}$
5. $\mathcal{P}_{De}^L = \{(E_e, e.com/)_1, (A_e, a.com/)_2\}$
6. $\mathcal{P}_{De}^A = \{(E_e, E_e, +*)_1, (A_e, E_e, (+r, +w))_2\}$

Figure 4.8: Example Policy

authorized state by using cookies. The server returns the cookie as an authorization token, and the browser attaches the cookie in the next HTTP request to show that the request is coming from an authorized user. However, since the browser automatically attaches the cookie to the HTTP requests for the target server, the cookie will be attached to all requests, including those issued by malicious attacker's content. CSRF allows an attacker to issue illegal commands to innocent servers by "riding" on the existing au-

thorized session. Other browser-managed authorization mechanisms such as the HTTP authorization header and client-side SSL authentication are also vulnerable to CSRF.

Unlike XSS, CSRF does not require a malicious script to be injected into the content on an innocent server. CSRF can not only compromise the authenticity of the commands sent to the server, but also be used to compromise the confidentiality of information protected by authentication; when an HTTP response returns content in the JSON format [28], an attacker may steal information by overriding the object constructors [26].

There are two typical countermeasures for CSRF. First, the Web server can check the `Referrer` header of the HTTP request, which should indicates the URL of the content originating the request. However, some browsers allow disabling the Referrer header in order to protect users' privacy, and thus this is not an always-reliable method. Second, an application can authenticate an HTTP request by using a secondary security token in the request parameter, in addition to the cookie. The secondary token insures that the request is originating from a genuine part of the Web application, which includes the token in the content (e.g., in a hidden input field). However, many Web applications do not implement these countermeasures correctly, and thus it is important that users can protect themselves from such vulnerable applications.

Our Approach for Preventing CSRF

In order to prevent CSRF, we introduce the notion of *automatic cookie (c)* permission in our browser model. For example, in the example environment

of Fig. 4.6, the server `a.com` is a potential target of CSRF attacks by `e.com`. Therefore `a.com`'s policy defines that the *automatic cookie (c)* permission for the server `a.com` is given only to the content with the security label A_a.

Conversely, `a.com` may maliciously try to carry out a CSRF attack against `e.com` with the JavaScript code as follows.

```
document.imgs[0].src=
'http://e.com/x?cmd=doSomething'
```

This code will trigger an HTTP request to `e.com` with the given URL. In the current browser, the request is also associated with the document cookie from `e.com`. However, in our proposed model, this JavaScript code is part of the content downloaded from `a.com`, and associated with the data label A_e in the `e.com`'s policy (policy 5-2). Since the content with the data label $\top$ does not have the cookie (c) permission in the access control policy of `e.com` (policy 6), the cookie of `e.com` will not be sent along with the HTTP request.

4.6.3 Local Network Attacks

In browser-based port scanning attacks, malicious JavaScript attempts to access local IP addresses in the user's network. By detecting the presence of default icon images of specific Web servers or from the type of errors (e.g., HTTP 404 Not Found or connection errors), the attacker can detect the presence of hosts and even determine the types of the Web servers. An attacker can also send commands to networked devices or services. For example, an attacker may use JavaScript to send arbitrary print commands to a networked printer [94], or to configure a victim's home broadband router to change its DNS settings [86].

Since JavaScript is executed on the user's browser after being transferred through an ordinary HTTP port, an attacker can easily attack the user's local network behind a firewall. An attacker may also combine these techniques, e.g., finding potential victim servers by port scanning, and then send illegal commands to them, while leveraging CSRF to bypass the user authorization.

Our Approach for Preventing Local Network Attacks

Fig. 4.9 shows an example of JavaScript-based port scanning. In this example, the JavaScript code downloaded from `e.com` tries to access the given port of the server by dynamically modifying the src attribute of an `<img>` element, and determines the presence of the host from the presence of images and the type of errors.

This attack cannot be prevented by the same-origin policy, because it only controls the network access by the XMLHttpRequest objects.

In our proposed model, all network accesses, including those that use HTML elements and attributes are also the target of the access control. For example, in the example environment in Fig. 4.6, the user's access control policy is defined such that only content from `u.com` can access URLs on `u.com`.

For example, assume that `e.com/e.js` includes script shown in Fig. 4.9. The script will be associated with the data label $\{\top_u, E_a, E_s\}$; since the user's policy define no labels for `e.com`, it will be $\top_u$ in the context of the user defined policy. Since the content with the data label $\top_u$ does not have any permission in the access control policy 2, the browser does not issue HTTP requests to `u.com`, because of the lack of the access permission of.

```
function portScan(callback, host, port) {
  var timeout = 100;
  var img = new Image();
  img.onload = function () {
    callback(host, port, 'open');
  }; img.onerror = img.onload;
  setTimeout(function () {
    callback(host, port, 'closed');
  }, timeout);
  img.src = 'http://' + host + ':' + port;
};
```

Figure 4.9: Port Scanning

4.7 Discussion

We chose XPath to identify parts of document that would include content with different trust levels, and to associate the access control policy with the subdocument. However, because of the flexibility of HTML and the client-side JavaScript, XPath is not an ideal mechanism for specifying the sub document.

For example, a mashup server may try to confine the content from a third-party `e.com` within a `<div>` element, e.g., `<div id='e'>`*`content from e.com`*`</div>`. However, `e.com` can change the document structure by inserting a fake close tag. For example, if the content from `e.com` includes a close tag `</div>`, then the label `e` will not be associated with content after this close tag. Such problem can be avoided by good programming practice, i.e., to insert third-party content using JavaScript and DOM API.

Likewise, `document.write(`s`)` can change the document structure when string s includes a close tag. This happens because the current browsers use the *early evaluation* approach, i.e., `document.write()` is executed while the browser parses and renders the main HTML document. Such behavior has significant impact in the secure browser architecture, since it allows an at-

tacker to circumvent the access control policy. In order to enforce the labeling policy on the inserted content, we changed the behavior of `document.write` to *late evaluation*, i.e., the content will be inserted after the DOM of the main document is constructed, and thus does not change the document structure.

This chapter did not discuss how to attach policies to the contents, but we assume that the content policy is sent on the same HTTP response as the content itself, most likely in the form of an HTTP header. It is important to make sure that the policy cannot be modified by the client-side JavaScript.

There are several prior work propose finer granular mechanism of binding the *domain* information to each page, which summary is given in [56]. This proposal chose a simple approach of distinguishing finer grained domains by URL path prefix, but it is possible to adopt other approaches, such as the server-side certificate, with some extension to the labeling policy in the proposed model.

4.7.1 Challenges

Several technical challenges need to be solved for the proposed model to be implemented.

First, the JavaScript engine needs to be extended to support fine-grained dynamic information flow tracking. Such an extension might easily introduce a large performance overhead, and thus an effective implementation is necessary to insure the browser remains usable in real-life applications.

Second, as mentioned above, a policy attachment mechanism has to be designed and implemented.

Third, the Web browser has to be modified so that every access to re-

sources will be mediated by a reference monitor that enforces the access control policy based on the proposed security model.

Forth, the current model omitted support of implicit flows because it has little effect when assuming no conspiracy between trusted and untrusted content. Support of implicit flows will require further research due to the dynamic nature of JavaScript and the need for effective un-tainting or re-labeling.

Finally, various details of the browser behavior (e.g., SSL) have been omitted to keep the model simple. In addition, support of privilege overriding in a trusted code (similar to `doPrivileged` in Java) will also be needed in real-life applications. An effective and performable implementation is another topic that requires further research.

Chapter 5

Dynamic Information Flow Control Architecture

This chapter describes Dynamic Information Flow Control Architecture (DIFCA), which allows tracking and controlling fine-grained information flow on the server-side Web application programs.

5.1 Introduction

In a typical three-tier Web application server (Figure 5.1), sensitive information, such as user's personal information or credit card numbers, is stored in a database. Access to the database is controlled at the database management system, and limited to only authorized users. However, such control is not sufficient to protect sensitive information.

First, after the application retrieves sensitive information from the database, it is up to the application code how to handle such information. An erroneous application may carelessly release such information to an undesirable desti-

nation. Furthermore, a single database table may contain data that belong to different classification levels. Some database systems support fine granular access control, but again, once the data is retrieved from the database, protection of the data depends on the application behavior.

Second, many of today's Web applications use a single database account for processing requests from multiple Web users, in order to effectively reuse database connections through connection pooling to optimize performance. The access control at the database is not effective when the same database account is used for all users. For example, a credit card number for a user A may be presented to user B due to a bug in the application, because the single database account cannot distinguish between users.

Figure 5.2 shows a sample application in which a program bug can cause undesirable information flows. This is a simple example of an on-line shop servlet, which receives a user name and the item name from an HTTP request, and processes purchase request using the user's credit card number stored in the database. The `processPurchase` method checks the validity of the credit card number, and stops process by returning false when any problem is detected (e.g., the credit card is expired). We assume that the information received from the user via an HTTP request is not confidential, while credit card numbers in the database are confidential. (We assume that a proper user authentication takes place in advance, and the communication channel is protected by SSL or TLS, and thus the information in the HTTP request can be trusted.)

When looking at this program from the aspect of the confidentiality of the credit card number, it has two problems.

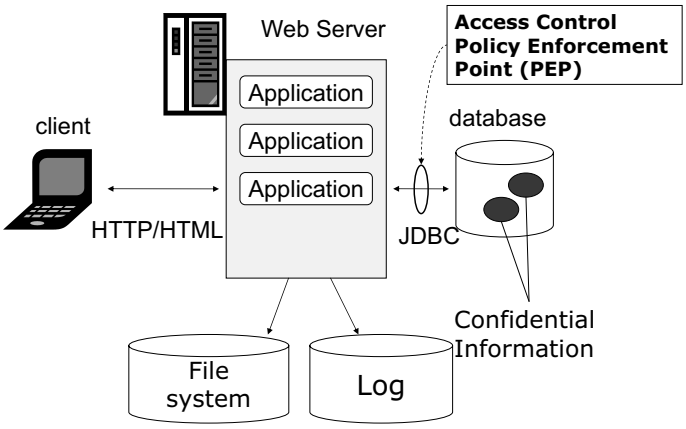


Figure 5.1: Three-tier Web App. Example

```
protected void doGet(HttpServletRequest req,
                     HttpServletResponse res) throws ... {
    String user = req.getParameter("user");
    String item = req.getParameter("item");
    PrintWriter pw = new PrintWriter(res.getOutputStream());
    ...
    String credit = getCreditCardInfoFromDB(user);
    boolean b = processPurchase(user, item, credit);
    if(b){ // succeeds
        pw.println("Purchase Succeeded: <br/>");
        pw.println("Name: " + user + "<br/>");
        pw.println("Item: " + item + "<br/>");
        pw.println("Credit Card: " + credit );
    }else{ // failed
        printlog("Invalid credit card: " + credit);
    }
    ...
}
```

Figure 5.2: Problematic Servlet Code Example

1. When the `processPurchase` method succeeds, the application sends back the credit card number to the user via an HTTP response. Although the credit card number belongs to the user, it should not be sent to the user unless necessary since the number may be peeped at over the user's shoulder, or leaked from the browser cache.

2. When the `processPurchase` method fails, the application outputs the user's credit card number into a log file. Confidential information should not be output into log files unless necessary.

We propose a system which prevents such undesirable information flow. That is, in the above example, the system detects undesirable information flow and stops processing when a credit card number is being output. In addition, when data is properly sanitized (e.g., when a credit card number is masked), the data should be "declassified" to indicate that it is no longer confidential.

In order to achieve fine-grained information flow control, language-based information flow control is receiving attention [84]. Much of the past research focuses on static analysis of information flow in a program using the type system or data flow analysis. However, we consider the practical use of such technologies to be difficult, because 1) It is difficult to analyze complicated data structures and control flows in multi-threaded object-oriented languages, 2) When a programing language is extended to include security functionality, existing software resources, such as libraries and development tools cannot be reused without adaptations, and 3) static analysis cannot take the dynamically generated code into account.

A dynamic information flow control approach has been proposed by Hal-

dar, Chandra and Franz [46] [47] [35] to take advantage of the rich state information from a running application. They use a bytecode rewriting technique to modify the application bytecode to insert extra code for tracking the information flow of an application. Their approach tracks information propagation through method invocations and field access. We propose a Dynamic Information Flow Control Architecture (DIFCA), inspired by the Haldar et al. approach. DIFCA inherits characteristics of being able to support dynamic conditions of running applications, not requiring any sourcecode from the target software, and being independent of Java VM implementations. In addition, in our system: 1) information flow is tracked and controlled at the granularity of primitive data types through most of the JVM instructions including logical and arithmetic computations, the operand stack and local variables operations, method invocation, and exceptions. 2) it effectively labels data for input or output from or to external environments, especially data exchanged with databases, and 3) it supports fine-grained application-level policies, including declassification policies. A more comparison with Haldar's approach is discussed in Section 3.

We implemented DIFCA on top of Apache Tomcat, and integrated Application Privacy Monitoring for JDBC (APM4JDBC) to enforce security policies in database (See Section 5.4).

The rest of this chapter is organized as follows. Section 5.2 gives an overview of DIFCA and its basic concepts. Section 5.3 shows the architecture of DIFCA, and its detailed method for information flow control in execution of Java bytecode instructions. Section 5.4 describes integration of the databases for fine- grained information flow control between the database

and the application server. Section 5.5 describes the prototype implementation.

5.2 Overview of DIFCA

We propose a Dynamic Information Flow Control Architecture for Java (DIFCA) to control the information flow of Java applications.

DIFCA allows administrators to define security labels for external resources (such as files, network and databases) as well as the information-flow policies between labels. During the execution of applications, DIFCA keeps track of propagation of security labels for the data, and detects any output that violates the policies.

The run-time functionality is inserted into the application bytecode as inline reference monitors (IRM) [33], using the bytecode rewriting technique. When the application is executed, the inserted IRM code communicates with the Access Control Module (ACM), to notify it of the state of the running application. The ACM is implemented as a Java class, and a unique instance is created and associated with each thread. An ACM has an internal structure similar to a Java Virtual Machine (JVM). However, instead of holding data, an ACM holds security labels that are associated with data in the JVM. The ACM propagates security labels of data by synchronizing itself with the code execution of the JVM.

The ACM takes two sets of policies: the labeling policies, and the information-flow policies. Since the instrumented bytecode is independent of the policies, the policies can be late-bound to the application at run-time.

5.2.1 Labeling Policies.

In DIFCA, any input or output to external resources is identified by Java APIs associated with the operation, that is represented in a form of pseudo-URI such as "java:*class_name.method_name*". Some resources require finer granular control for labels than APIs; e.g., the labels of files and network resources need to be identified by their locations, rather than by APIs. Therefore, such location are represented in the form of a URL.

In addition, a resource is either structured or unstructured, with regard to the labeling of its information. Examples of unstrucutred data are plain text files, where each file includes information with the same confidentiality. A database system is an example of a structured resource. Each datum in a database needs a more dedicated labeling policy, since each table may include columns with different classifications, and the classification may depend on the context of the query. DIFCA leverages Application Privacy Monitoring for JDBC (APM4JDBC) [4] to allow dynamic labeling of the database query results.

5.2.2 Information Flow Policies.

DIFCA is policy agnostic and thus can flexibly adopt different types of policies, such as Biba [24], Bell-LaPadula [22][65], or the lattice model [30].

For the sake of simplicity, in the following example we use a simple Bell-LaPadula-type policy which has only two labels and ; i.e., information with the label cannot flow into , while information with the label can flow into . Labels are propagated when information flow occurs from explicitly labeled data.

5.2.3 Label Composition.

When a value is derived from the composition of two values with different labels, the label of the resulting value becomes the composition of the two labels of the original values. Here, the composed label should be the Least Upper Bound (LUB) of the two labels; i.e., the lowest label which satisfy both of the two labels. For example, when $a + b = c$ where a is and b is , the label of c needs to be . This is because when an attacker learns the values of both b and c, he can easily infer the value of a.

5.3 DIFCA Architecture

Figure 5.3 shows the architecture of DIFCA. IRMWriter takes the bytecode of an application as input, and inserts inline reference monitor (IRM) code into it. This instrumentation process may happen either before application deployment, or on-the-fly when the code is loaded by a class loader.

A JVM has a JVM stack js^t for each thread t, which is a stack of frames, where each frame (fr_i^t, where i denotes the position on js^t) holds an operand stack os_i^t and a list of local variables lv_i^t for the method m_i^t that is being executed. When a new method m_{i+1}^t is called, a new frame fr_{i+1}^t is created and pushed onto js^t. Likewise, an ACM has a stack structure $l(js_t)$ corresponding to the JVM stack for each thread. Each $l(js_t)$ is a stack of frames-for-labels $l(fr_i^t)$. Instead of holding data operated upon by a method, each $l(fr_i^t)$ holds the labels of local variables $l(lv_i^t)$ and the operand stack $l(os_i^t)$ associated with the data.

As each bytecode instruction is executed in the application, the IRM code

synchronizes the state of the ACM with the state of JVM so that the labels in $l(js^t)$ represents the security label of the data in js^t being operated upon in the application.

In a JVM, object instances and static field data are stored in the heap area. The ACM has three tables for holding the labels for the objects in the heap: the object label table (OLT) for the labels of objects and the fields of the objects, the array label table (ALT) for the labels of array elements, and the class label table (CLT) for the static fields of the classes.

A JVM also has a method area and a constant area for holding the bytecode of methods as well as the constant data of Java classes and interfaces. We regard these areas as non-confidential, and associate them with the special label by default (i.e., $LUB(,l) = l$ for any given label l, and either or can flow to destination with the label).

5.3.1 Information Flow in Java Bytecode

Programs written in the Java language are compiled into Java bytecode, a standard pseudo-machine language that is executed on a JVM [68]. Since we attempt to support applications with no source code, we target the Java bytecode to track the information flow of an application.

The JVM Specification [68] defines about 200 instructions, but in many cases a single semantic operation is defined in multiple instructions for different data types. Similarly, most instructions that concern local variables have variants that include frequently used local variable indices

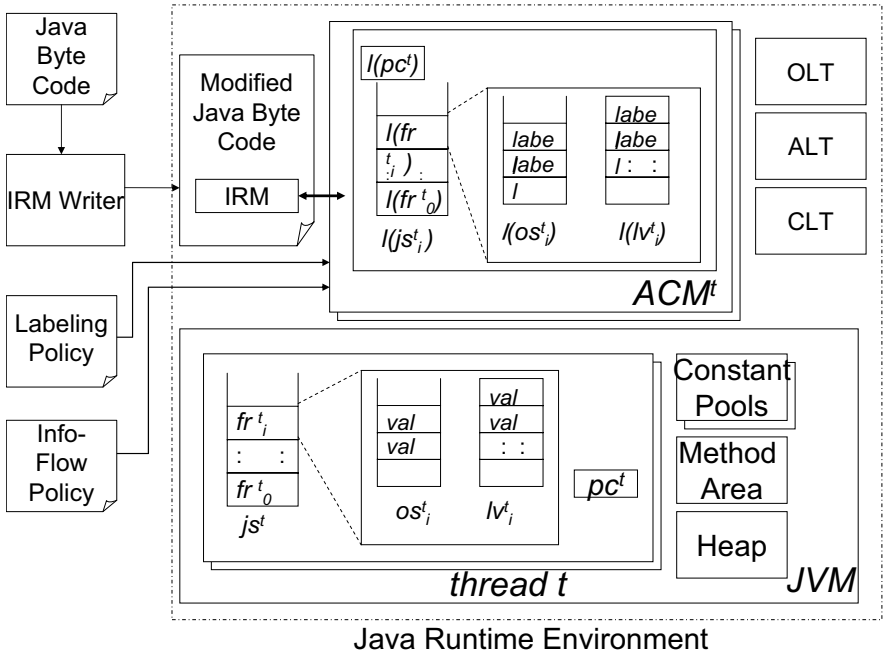


Figure 5.3: DIFCA Architecture

5.3.2 Stack and Local Variable Operations

When information is exchanged between the operand stack os_i^t and the local variable table lv_i^t, e.g., by a `LOAD` or `STORE` instruction, the ACM propagates the label between $l(os_i^t)$ and $l(lv_i^t)$. When two values are combined to create a new value, e.g., as a result of a binary operation, the ACM obtains the composition of the two labels and associates it with the new value.

When a constant value is loaded onto os_i^t, the label is associated with the value (unless the load operation was affected by an implicit flow as discussed later in Section 5.3.5).

For example, Figure 5.4(a) is a simple Java program which adds the

variable b and the constant 1 and assign the result into the variable a. Figure 5.4(b) shows the bytecode representation of the same program.

The ACM follows the operations of the JVM. For example, the ACM first loads the labels of b from $l(lv_i^t)$ to $l(os_i^t)$ at the `LOAD` operation and then the label of 1 (`NONE`) to $l(os_i^t)$. When the `ADD` operation is executed, ACM obtains the composition of the two labels on $l(os_i^t)$, in this case $newlabel = LUB(l(os_i^t[j-1]), l(os_i^t)[j])$, where j denotes the highest position of $l(os_i^t)$ as of the time the operation is performed. Then the resulting $newlabel$ is pushed onto $l(os_i^t)$. Finally, the $newlabel$ is propagated to $l(lv_i^t)$ to be associated with the variable a at the `ISTORE_1` operation.

5.3.3 Object and Field Access

Each object can be associated with a security label, but each field of an object may have different labels than the object itself. Therefore, our approach is designed to handle them separately.

A JVM stores objects into the heap area, and accesses to the fields are done through the `GETFIELD` and `PUTFIELD` instructions. The ACM stores the labels of the objects and their fields in the Object Label Table (OLT). The labels are propagated between OLT and os_i^t at each `GETFIELD` or `PUTFIELD` instruction. Similarly, the labels of static fields and the labels of array elements are managed in the Class Label Table (CLT) and the Array Label Table (ALT), respectively.

5.3.4 Method Invocation

When a method is invoked, information is propagated between the caller method and the callee method through method arguments and the return value.

When a method `foo()` invokes the method `bar()`, the method arguments are the output and the return value is the input, from the view point of `foo()`. In contrast, for `bar()`, the method arguments are the input and the return value is the output. Therefore, DIFCA allows defining the labeling policy of input and output for `foo()` and `bar()` separately. When an explicit policy is defined for an input to a method, then a security label is associated with the input data. When an explicit labeling policy is defined for the method output, then the label of the output data is compared with the label of the method, and execution is stopped when any violation of the information flow policy is detected. For example, when some data with label is being specified as an argument for a method with the output label , that invocation causes an information-flow violation.

It should be noted that both the caller `foo()` and the callee `bar()` are not necessarily instrumented with IRM. The IRM Writer can only instrument the application code, and the standard Java libraries and the middleware (e.g., Web server) must not be modified. If the standard libraries are also IRM-enabled, the IRM code would recursively call the IRM code and the code could not be executed properly. Therefore, when `foo()` invokes `bar()`, there are four possibilities with regards to their IRM enablement: 1) both of `foo()` and `bar()` are IRM enabled, 2) only `foo()` is IRM enabled, 3) only `bar()` is IRM enabled, or 4) neither `foo()` nor `bar()` is IRM enabled.

In case 1), the label of the arguments is explicitly propagated by IRM by copying the labels from caller method's $l(os_i^t)$ to the callee method's $l(lv_{i+1}^t)$, while the arguments are copied from the os_i^t to lv_{i+1}^t. When returning from the method by the `RETURN` instruction, the label of the return value is popped from $l(os_{i+1}^t)$ and pushed to $l(os_i^t)$.

In case 2), since only m_i^t is IRM-enabled, the `INVOKE` instruction is IRM-enabled, but the `RETURN` instruction that is executed from m_{i+1}^t is not IRM-enabled. Therefore, the caller method m_i^t infers the label of the return value from the composition of the labels of the input arguments and the target object itself.

In case 3), since the caller method is not IRM-enabled, the callee does not receive the label of the input values unless an explicit policy is specified and thus the label is associated with the input argument by default.

5.3.5 Implicit Flow

Even when no explicit flow occurs from to , the information can be infered from the information when the information affects the control flow of the program. For example, in the example code in Figure 5.5(a), the value of x can be inferred from the value of y after the `if` statement, since the value of y differs by the value of x. Such information flow is called an implicit flow [31].

In order to detect such an implicit flow, DIFCA associates a security label with the program counter to show any implicit flow caused by the execution of the code. Let pc^t be the program counter of the thread t and $l(pc^t)$ be the label associated with pc^t. In the above example, when the value of x causes

a conditional branch in Line 3, $l(pc^t)$ is raised to . Then when the value is assigned to y, the label is propagated to the label of y.

Figure 5.5(b) shows the Java bytecode that is compiled from the Java source code in Figure 5.5(a). In Line 6 in the bytecode, `IF_ICMPNE` evaluates the value on the os_i^t to branch conditionally. Lines 9 and 10 are the code that are executed conditionally depending on the two values that are on os_i^t (i.e., x and the constant 1). All of the values that are affected by these conditions need to be associated with the security label of the composition of the original value and the program counter. In the above example, when `IF_ICMPNE` is executed, $l(pc^t)$ becomes the composition of the label of x and the constant 1 (i.e.,), and then in Line 9, the value propagated by the `ISTORE` instruction will be $LUB(l(os_i^t[j]), l(pc^t))$, which is the composition of the label of the value on the operand stack $l(os_i^t)$ and the label $l(pc^t)$.

Similarly, all of the IRM operations described before, need to compose $l(pc^t)$ in addition to the label of the operands. For example, when a constant value is loaded onto OS, the label $l = LUB(, l(pc^t))$ is actually pushed onto $l(os_i^t)$. When multiple conditional branches are nesting, $l(pc^t)$ evolves to reflect the context; i.e., each time the pc^t reaches a new branch, $l(pc^t)$ is updated to be $LUB(l(pc^t), l(c))$ where $l(c)$ denotes the label of the branch condition. $l(pc^t)$ is reset at the join point of each conditional statement.

Strictly speaking, the implicit flow occurs whether or not the body of the `if` statement is executed; that is, one can infer that the value x is not 1 when the value of y is 0. It should be noted that a purely dynamic approach, can propagate the label only when the assignment is done, when there is no knowledge of other possible execution paths [43][85].

```
                        0: ICONST_1
                        1: ISTORE_1
                        2: ICONST_2
                        3: ISTORE_2
                        4: ILOAD_2
1: int a = 1;           5: ICONST_1
2: int b = 2;           6: IADD
3: a = b + 1;           7: ISTORE_1
```

(a) Java Program (b) Bytecode

Figure 5.4: Code Example for Addition

```
                        0: ICONST_1
                        1: ISTORE_1 // x
                        2: ICONST_0
                        3: ISTORE_2 // y
                        4: ILOAD_1
1: x = 1; // HIGH       5: ICONST_1
2: y = 0; // LOW        6: IF_ICMPNE #11
3: if(x == 1){          9: ICONST_1
4:   y = 1;            10: ISTORE_2 // y=1
5: }                   11: ...
```

(a) Java Program (b) Bytecode

Figure 5.5: Code Example for Branch

5.3.6 Exceptions

An exception in a Java program is caused by either 1) the JVM (e.g, division by zero), 2) a `throw` statement in the library code, or 3) a `throw` statement in the application code. In DIFCA, exceptions explicitly thrown by applications will be assigned the security label that is determined from the label of the data that is set in the exception as well as $l(pc^t)$ of the code that throws the exception. When an exception is not caught, the frames are popped in the

JVM stack. In order to synchronize the stack depth, the IRM inserts default exception handlers to capture the exception, to synchronize the state of the ACM, and to throw the exception immediately.

When an exception is directly thrown by the JVM, the IRM's ability to detect the label of the exception object is limited. For example, when `division by zero` is reported by java.lang.ArithmeticException, it can be inferred that the dividision operand was 0. However, since this exception is caused by the JVM itself, it is difficult for the IRM to reflect the label of the operand to the label of the exception.

5.3.7 Multi-Threading

The types of information flowing between threads in a multi-thread program can be classified into 4 types: 1) Initialization parameters of the child thread objects that are set by the parent thread, 2) Pairs of threads communicatiing through shared global objects (e.g., singleton objects), 3) Information exchanged via external resources such as files, databases, or system properties, or 4) Covert communication channels that make use of Java's multi-thread capabilities, such as thread synchronization and interruption.

In DIFCA, information flowing through 1) and 2) is captured since the labels of the objects in the heap area are managed by the global tables OLT, CLT, and ALT. Information exchanges through external resources can be captured as long as all of the resources are labeled properly.

5.3.8 Declassification

Declassification is an important issue for the practical use of an information flow control system to mitigate the label creeping problem [31]. For example, in a system that implements the Bell-LaPadula model with two labels and , all processes that can read from both of and information must not have a write permission to the information. Therefore, as information propagates, information that originally had the label tends to be associated with the label. Since each process is regarded as a black-box for the operating system level information flow control, it is difficult to avoid the label-creeping problem.

Even in a language-level information flow control system, the labels of data tend to become more strict as the program is executed. In particular, this trend is significant when the label of the program counter is composed into the labels of the variables to capture the implicit flows.

However in reality, not all information produced from confidential information is confidential. For example, credit card numbers are usually regarded as confidential, but it is a common practice to mask the credit card number except for the last 4 digits to make it public information that can be printed on the bill, e.g., `"****-****-****-1234"`. Another example is a password. The password itself is confidential but its hash value or a boolean result of authentication, both derived from the password, are public information.

DIFCA supports declassification through a API-level declassification policy specification. For example, if there is an API method `String mask(String creditcard)` that masks a credit number except for the last 4 digits, the labeling policy can be defined to force the return value of this method to , and thus allows declassification without modifying the code.

5.4 Labeling on Database Queries

JDBC (Java DataBase Connectivity) is a standard API for accessing databases from Java applications. The standard API encapsulates the complexity of each database management system and its driver, and allows Java applications to utilize databases without concerns about the implementation- specific differences. A typical Java application connects to a database using JDBC, "logs-in" with a user account registered with the database, and issues queries using SQL.

When the database management system employs an access control mechanism, the access permission is linked to the given database user ID. However, many of today's Web applications use a single database account for processing requests from multiple Web users, in order to effectively reuse database connections through connection pooling to optimize performance. Therefore, the access control at the database access point is ignored.

Application Privacy Monitoring for JDBC (APM4JDBC) [4] is a generic framework for a JDBC to intercept JDBC API calls and to insert customized behavior for each activation of the JDBC. Such behavior includes application-level access control for database queries as well as for recording database accesses for auditing purposes.

Figure 5.6 shows the architecture of APM4JDBC. Any query context (such as a web user account that is different from the database user account) can be corrected by the Context Handler, so that such information can be later utilized by the plug-in access controllers to filter and modify the SQL queries and responses.

DIFCA uses the APM4JDBC framework to collect the contexts of the

database queries, and labels the retrieved data based on the policies and the contexts of the queries. For example, DIFCA allows putting different security labels for each column on the queried data (e.g., associate the label with the credit card number in our example scenario introduced in Section 5.1). Similarly, DIFCA allows controlling data output into the database (e.g., make sure that application will not write a value with the label into a database column which should only hold the values). In addition, we can extend the labeling system to a richer model, such as the lattice model, and associate different labels for each user's transactions, and prevent contamination of information belonging to different users.

5.5 Prototype Implementation

DIFCA was prototyped on top of the Apache Tomcat Web container with JVM 1.5. The IRM Writer was implemented with Apache Byte Code Engineering Library (BCEL) [3], an open source toolkit for analyzing and modifying arbitrary Java bytecode. We modified the Web Application class loader in Tomcat to instrument only the bytecode of application classes when they are being loaded. The ACM is implemented as a singleton Java object.

Figure 5.7 shows an example of IRM-enabled bytecode after instrumentation. Lines with underlines are inserted IRM code that calls the ACM by invoking the methods of the ACM.

Limitations of the current prototype are that it does not support exceptions, and some policies (i.e., information flow policy and the database policies) are hard wired in Java code.

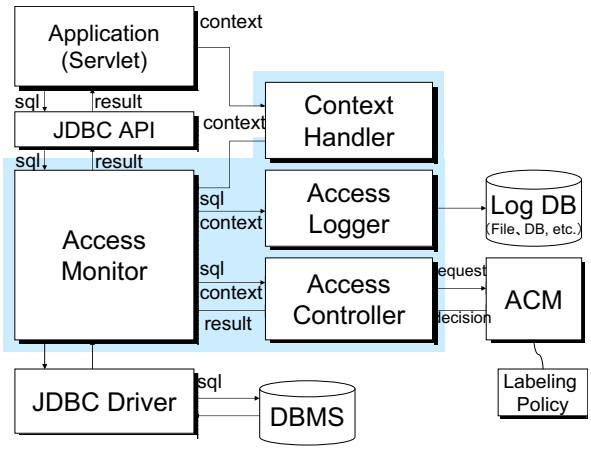


Figure 5.6: APM4JDBC

```
...
iconst_0
invokestatic acm.ACM.loadSingleVar (I)V
aload_0
invokestatic acm.ACM.pushSingleConst ()V
iconst_0
dup2
invokestatic acm.ACM.loadSingleArray ···
aaload
iconst_1
invokestatic acm.ACM.storeSingleVar (I)V
astore_1
iconst_0
invokestatic acm.ACM.loadSingleVar (I)V
aload_0
invokestatic acm.ACM.pushSingleConst ()V
...
```

Figure 5.7: IRM Enabled Bytecode

5.5.1 Policy Definition

The sample information flow policy is defined in a Java class which provides label comparison and composition as methods. The sample labeling policy (Fig. 5.8) defines the label for the HTTP requests and responses, and to the `printlog()` method. The policy on the `mask()` method defines the declassification policy on the masked credit card numbers.

DIFCA requires labeling policy only on API that concern input and out-

```
<Policy>
 <InputRule><Label>LOW</Label><Type>argument</Type>
   <URI>java:foo.shop.Purchase.doGet</URI></InputRule>
 <InputRule> <Label>LOW</Label><Type>return</Type>
   <URI>java:foo.shop.Purchase.mask</URI></InputRule>
 <OutputRule><Label>LOW</Label><Type>argument</Type>
   <URI>java:foo.shop.Purchase.printlog</URI></OutputRule>
</Policy>
```

Figure 5.8: Example Policy

put of data. Therefore, the administrator's burden of policy definition is smaller than security enhanced language such as Jif [76]. E.g., only 4 entries of labeling policy is required in example application in Section 5.1; other methods that does not concern with input and output of data will just propagate the label.

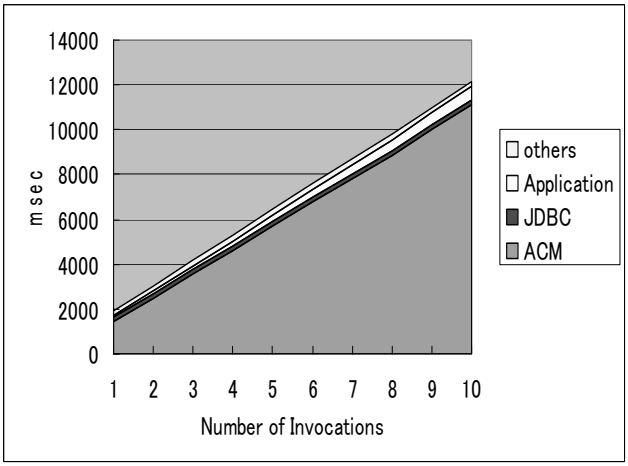
When no explicit policy is defined, DIFCA infers that a label of the value returned from a standard library method is the composition of the labels of the target object and the arguments. When this inference rule fails, explicit labeling policies need to be defined for such API. However, it is possible to pre-define a set of policies for each standard library and deploy with DIFCA, in order to mitigate administrator's burden to define them by themselves.

The policy on the database, which associates the label with credit card numbers, is hard-coded in a Java class in the current prototype. But it is obvious that we can extend the system to allow more flexible policy definition. Since the example policy adopts the simplest two-level labels, no context information was used for labeling the database query results. However, the architecture is policy agnostic and we can easily extend the policy to accommodate finer-grained policies in more flexible way; e.g., the credit card numbers that belong to different users are associated with different labels that corresponds to the user identities.

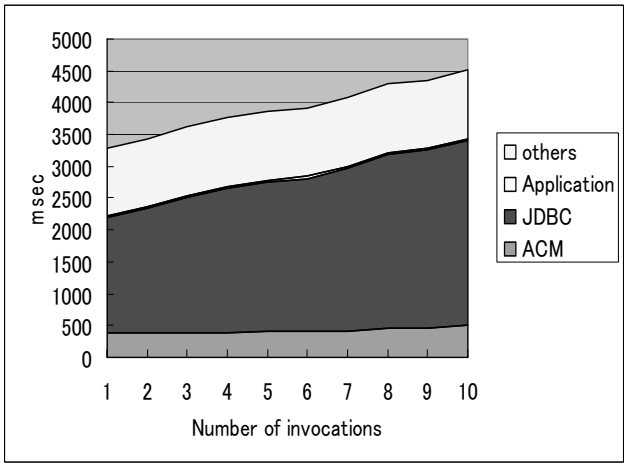
5.5.2 Performance Evaluation

We measured performance overhead of DIFCA in two types of Web applications, 1) only arithmetic operations without JDBC (Fig. 5.9(a)), and 2) sample application in Section 5.1 which uses JDBC (Fig. 5.9(b)). We ran each application for 10 times, and measured accumulated time consumption of each Java package using a profiler. The Web server (Apache Tomcat) and the MySQL database were set up on the same computer. In 1), the pure overhead by ACM over the original application class is about 18-24 times. In 2), JDBC connection and query consumes more than 90% of the entire execution time (even if excluding the connection establishment which occurs only once), and thus the overhead by ACM stays around 10%. But the pure overhead is about 13-18 times. IRMWriter instruments the bytecode only once when the application is loaded, and thus most of the overhead is resulted from ACM. Note that the pure overhead means the execution time comparison between the original application class itself and ACM, and the overhead in the turn-around-time of the Web application as a whole is smaller, especially when JDBC is involved. After instrumentation, each class file increases its size about 2-2.8 times.

In conclusion, although performance overhead caused by ACM is not small, we think that it is still acceptable in typical Web applications which uses JDBC.



(a) Arithmetic Operations Web App.



(b) JDBC Web App.

Figure 5.9: Performance Evaluation

5.6 Discussion

DIFCA enforces the language-based information flow control policies for Java applications. We use a bytecode rewriting technique to insert inline reference monitors (IRMs), and thus 1) the IRMs can utilize the detailed conditions of the running applications, 2) it does not require source code of the target application, and 3) the system is independent of JVM implementations. DIFCA tracks the propagation of information in the program through most of the JVM instructions, and controls the input and output to the external environment based on the given information flow policies. DIFCA also intercepts the JDBC queries to effectively label the query results, and control input to and output from the database.

However, the current proposal still leaves gaps for future research. First, the purely dynamic approach can discover only information flows that are actually executed, and especially cannot detect all of the implicit flows. Second, the current approach requires a bytecode-level IRM to be inserted for every bytecode instruction of the original code, and causes significant performance overhead. Third, since *OLT* stores object references with associated security labels, it prevents target objects from garbage collected, and causes memory overhead at run-time. Fourth, terminating the transaction due to the information flow violation may cause problems in database consistency. It is inherently difficult to handle such exceptions without modifying the applications. Some of the problems may be acceptable when using DIFCA for the pre-deployment test, but care needs to be taken to define the test cases with good coverage.

Usability and policy specification is another challenge that needs to be

addressed. DIFCA does not require the source code of the target applications, but the policy writer still needs to understand the structure of the program and the semantics of the methods. Especially when a declassification policy is defined for a method, such a definition may easily introduce human error, unless the semantics of the method are well defined and the consequences of the declassification are well understood. This is a future topic for allowing easy and safe policy definitions without needing knowledge of the source code.

Chapter 6

WebDLP: Web-based Data Leakage Prevention

6.1 Introduction

In Software as a Service (SaaS), application software is hosted in a service provider's infrastructure and capabilities are provided to the customers as services. The cost of the software is billed as a utility. SaaS is gaining widespread acceptance because it can reduce the initial investments and the maintenance costs. With the emergence of SaaS, Web browsers have become generic middleware for running the client-side user interface of Web-based applications, such as Web-based e-mail, online chat, file-sharing, etc. As a result, SaaS and hosted applications increase the risk of the leakage of sensitive data from a Web browser to external servers, either because of users' mistakes or attacks by malicious adversaries.

Data leakage via Web traffic is difficult to detect or prevent for three reasons.

First, traditional perimeter defenses using firewalls cannot prevent data leakages via Web channels, because most firewalls allow HTTP connections from client machines in a private network to external servers so that internal users can use various external Web services. It is rarely realistic to prohibit all external accesses, because that would likely decrease user productivity even if it is more secure.

Second, it is difficult to distinguish between data flows to trusted SaaS providers and to non-trusted services by using a coarse-grained content-inspection approach. In many cases, a user or a company makes a contract with a trusted SaaS provider. This means some sensitive data can be sent to the trusted SaaS provider but not to non-trusted servers. The growing trend of mashups and service integration makes the situation more complicated, because even if a user trusts a service, the trusted service may integrate other third-party services that the user does not trust. We want to detect the risks of data leaks from a trusted service to these third-party services.

Third, traditional Data Leakage Prevention (DLP) technologies focus on detecting specific types of sensitive data such as Personally Identifiable Information (PII) or credit-card numbers, typically by using pattern matching and dictionaries matching. However, the types of sensitive data handled by knowledge workers are more diverse, such as business secrets or intellectual properties. It is often difficult or impossible to recognize such diverse classes of sensitive data. In addition, it is difficult to identify sensitive data due to the massive amounts of data exchanged over HTTP.

This chapter proposes *WebDLP*, a fine-grained application-level proxy to detect potential data leakage issues. In particular, the proposed mechanism

addresses three issues above with the following approaches.

- WebDLP performs fine-grained analysis of the HTTP protocol and the Web content, such as HTML, JavaScript, XML and JSON, in order to determine the data flows in the Web traffic. In particular, it extracts the data elements from inbound and outbound Web traffic, and determines the actual data flows by comparing them with the traffic history. The method enables identification of potentially dangerous data flows among the massive amounts of communicated messages between the client-side JavaScript and the Web server.

- By recording the history of the Web traffic at the granularity of the data elements in association with their origins, WebDLP can detect data flows from one domain to another. This allows detecting cross-domain data flows that occur in client-side mashups, even though the user is not aware of such mashups.

- WebDLP can integrate any content classifiers to detect the sensitivity of the data flows on-the-fly. In this chapter, we describe a classifier that can be integrated with a document management system to help identify the leakage of known sensitive data. The proposed method measures the similarity of documents in a way that is robust against changes made by the users. For example, when some data is copied from a known sensitive document, that indicates a potential of high-risk of data leakage.

The rest of this paper is organized as follows. Section 6.2 describes the design principles and the architecture of WebDLP. Section 6.3 covers the

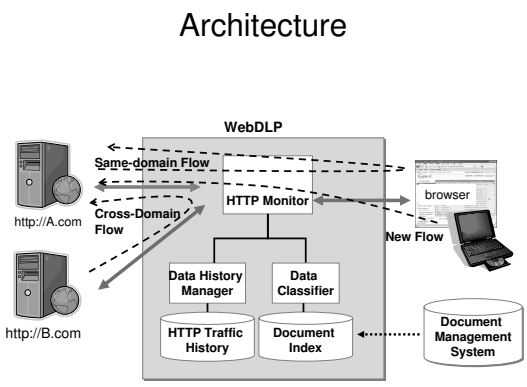


Figure 6.1: WebDLP Architecture

prototype implementation. Section 6.4 presents the result of our evaluation. Section 6.5 concludes this chapter and discusses our future research agenda.

6.2 Architecture

The overall architecture of WebDLP is shown in Figure 6.1. The architecture consists of three major components: *HTTP Monitor*, *Data History Manager*, and *Data Classifier*.

6.2.1 HTTP Monitor

The HTTP Monitor intercepts HTTP requests and responses to analyze the data in the Web traffic. The HTTP Monitor recognizes the two directions of data flow, *inbound* and *outbound*. The goal of the HTTP monitor is to prevent any leakage of sensitive data from the internal domain to external domains. In other words, preventing dangerous data leakages in the outbound data

flow is the objective of WebDLP. In this chapter we focus on the scenario to prevent sensitive data leakage from an internal client to external servers. Therefore, in the rest of this chapter, the HTTP requests sent by a client correspond to the outbound flows while the HTTP responses sent by the servers correspond to the inbound data flows.

The HTTP Monitor analyzes the HTTP protocol to extract the data that is transferred through it. For example, the request URLs, some client-controllable HTTP header values (such as Cookie headers), and the request and response bodies convey data. Each piece of data is analyzed according to its content type. For example, each request URL is parsed, and then the key-value pairs of the request parameters are extracted from it. Likewise, the HTTP Headers are parsed and the values (such as the cookie values) are extracted.

In addition, the HTTP Monitor analyzes the content that is conveyed in the HTTP request and response body, and then parses the content recursively until each piece of content is broken down into a set of atomic data elements. This is an effective method to extract the elements of the data from Web content that is often a mix-typed. For example, HTML files typically include JavaScript embedded in the `<script>...</script>` tags. Each piece of JavaScript code is parsed to extract data elements, such as constants, string literals, and object property names in the script. Also, a string literal in JavaScript code may include HTML fragments that will later be inserted into the document DOM tree by the JavaScript code. A similar recursive analysis is done for each of the standard Web-based content types, such as HTML, JavaScript, XML and JSON. Other content types (such as office

documents in a file-uploading HTTP POST request) are extracted as chunks and saved for later classification.

6.2.2 Data History Manager

The Data History Manager records and compares the inbound and outbound data flows at the granularity of the data elements extracted by the HTTP Monitor, determining the actual data flows, and more importantly, detecting the dangerous flows.

The data flows are categorized into three types by the Data History Manager:

Same-Domain Flows. In many cases, the client-side Web application extracts data in the inbound content, and sends it back to the server from which the content was received. (E.g., data flow from A.com to A.com via browser in Figure 6.1). A traditional example of this class of flow occurs with a static link in an HTML document, as well as an HTML form with hidden fields or default values. Modern Ajax applications often process data using the client-side JavaScript code that sends the data back to the server as asynchronous HTTP communications. Such same-domain flows are considered to be safe, because there is actually no real information flow from the client to the server.

Cross-Domain Flows. When some data included in the inbound content from one server is sent to another server via the client-side JavaScirpt code, this is considered to be a cross-domain data flow. (E.g., data flow from B.com to A.com via browser in Figure 6.1). The cross-domain flows create potential cross-domain data leakage risks, because these

Table 6.1: Example of Data Flows

Data Received	A.com	<script> var x="abc def";</script> <input name="ghi" value="jkl">
	B.com	{"X": 123, "Y": 456}
HTTP Traffic History	A.com	"x", "abc def", "ghi", "jkl"
	B.com	"X", 123, "Y", 456
Data Being Sent	Original HTTP Request	http://A.com/?ghi=jkl&X=123&Z=456&msg=hello+world
	Same-Domain Flow	"ghi", "jkl"
	Cross-Domain Flow	"X", 123, 456
	New Flow	"msg", "hello world"

flows might be the result of malicious intent, such as a malicious component in a mashup application or a cross-site scripting attack, which steals sensitive data from a trusted Web application and sends it to the attacker's server. However, due to the prevalence of mashups, there are many legitimate cross-domain flows in many of today's Web 2.0 applications.

New Flows. When some data is sent to a server and the data has not been observed in previous inbound flows, it is considered to be a new data flow. (E.g., data flow from the browser to A.com in Figure 6.1). New data flows may occur either as a result of user input or as a result of the execution of the client-side JavaScript code. For example, JavaScript code may generate a random value and send them to the server as a nonce.

An example of data flows is shown in Table 6.1.

Since the same-domain flows have low risks, the cross-domain flows and new flows are the targets of the WebDLP system to further assess the degree of data sensitivity by using content analysis technologies.

In addition, if the same data element is sent repeatedly, the Data History Manager ignores the redundant data elements. This design choice was made by observing the behavior of several Web applications. Our initial expectation was that the most of the new flows would be caused by user input data. However, the reality in many Ajax applications is that the majority of new data flows include values that are generated by the client-side JavaScript, such as by using random number generation or string concatination, that represent some unique identifiers (such as session or object identifiers) or fixed keywords (such as object property names). Since such data tends to be sent repeatedly, ignoring the redundant data helps reduce the volume of the extracted data flows.

6.2.3 Data Classifier

The Data Classifier analyzes the data elements, and determines their sensitivities. WebDLP can use any content classifiers on-the-fly to detect the sensitivity of the data flows, but in this chapter we focus on a similarity-based classifier that uses data similarity as the metric of document sensitivity.

Similarity-based Classifier

A key observation behind the similarity-based classifier is that knowledge workers often reuse documents and their contents to make a new document. For example, when preparing for a presentation, many people start by collecting existing documents or past presentation slides as reusable content, copy and paste the old content into the new presentation slides, and then modify them or add new content. The reused documents may originate with

other people, such as coworkers. Such document reuse often happens recursively, and content propagates within an organization, or even across multiple organizations.

When working in a project team, it is often the case that a document is exchanged between team members, and each time a new revision of the document is created by each team member who adds or modifies the content. Such data exchange and group editing is often the nature of the collaboration style of white collar workers.

One problem in document reuse is that the origin and the class of the document become unclear when the documents and content are propagated. For example, some organizations define internal rules to put a "Confidential" label on each sensitive document to indicate the document class. However, such classification labels may not be properly copied into the new documents that include content copied from a sensitive document. For example, in some presentation software, the footer and header information is lost when the document template is changed. When the document content is reused recursively, people are especially likely to lose track of the document origin, and easily become unaware of the sensitivity of the document.

We propose a DLP solution that is closely integrated with a document management system in an organization. Most organizations have some kind of document management system for sensitive documents, such as documents received from customers, intellectual properties, or business secrets. Such sensitive documents are often replicated on employee PCs for study or modification, but the original sensitive documents in the repository can be used to detect other revisions or even part of the documents that are under a risk

of exposure.

In our proposed architecture, the Document Repository represents a document management system that tracks documents along with their associated sensitivity classes. The document charactersitics are extracted from the repository, and used by the similarity-based classifier to determine the level of sensitivity in data flows.

6.3 Prototype Implementation

A prototype of WebDLP was implemented on top of WebScarab [79], an open source HTTP protocol monitor. Our HTTP Monitor is implemented as a plug-in module for WebScarab, which intercepts HTTP requests and responses, parses their data, and invokes the Data History Manager and the Data Classifier.

A prototype of the similarity-based classifier was built by using the TF-IDF algorithm [95] to compare the weighted term frequencies between data. In particular, the current implementation uses Apache Lucene [34], an open source search engine that implements TF-IDF. The new flows and cross-domain flows extracted by the Data History Manager is canonicalized (e.g., by converting URL encoding and entity references to corresponding characters), and parsed by using Lucene's built-in StandardAnalyzer to extract terms, to query the Lucene's search index.

In the proposed prototype, a local agent software which crawls the local file system, extracts text from office documents files, and then registers the text per each page as well as a corresponding document class with Lucene's search index. It is straightforward to extend this approach to more central-

ized document management systems.

6.4 Evaluation

The prototype of WebDLP was evaluated with two metrics.

First, an objective of the WebDLP is to analyze high-volume Web traffic to extract the most interesting portions of the data flows. For this metric, the system is more successful if the amount of the extracted outbound data flows ("New Flows" and "Cross-Domain Flows") are smaller compared to the size of the overall outbound data flows as well as the actual size of the user input data. Second, the WebDLP system is more successful if the level of the data leakage risks associated with the data class is determined properly, based on the similarities to the known data set.

6.4.1 Efficiency of Data Extraction

Table 6.2 shows the efficiency of data extraction for three popular Web applications, Gmail, Hotmail, and Twitter. In the evaluation of each application, 10 user input messages were posted from Firefox browser v.3.5. Each column of the table shows the number of bytes of these types:

1. The total size of actual user input messages, such as the e-mail subject, body, and recipient address.

2. The total size of the HTTP request messages.

3. The size of the data extracted by simple protocol analysis, whithout using the traffic history data.

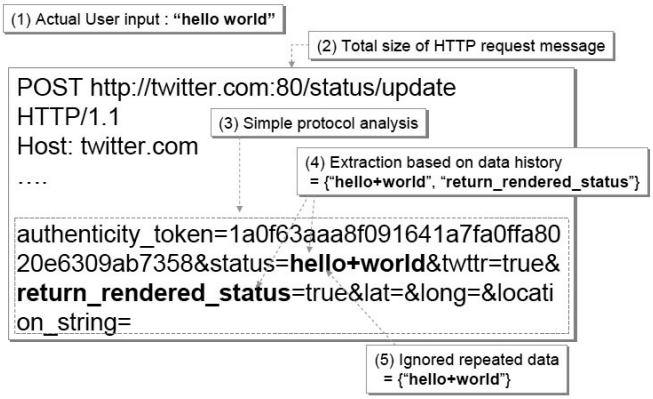


Figure 6.2: Example of Data Extraction

Table 6.2: Evaluation : Extracted Data Sizes (in bytes)

	1. user input	2. total HTTP req	3. simple analysis	4. extract by history	5. ignore repeated	5 vs 2	5 vs 3	5 vs 1
Gmail	819	123,666	4,758	1,847	962	0.73%	20.22%	117.5%
Hotmail	869	490,513	25,744	23,404	2,925	0.60%	11.36%	336.6%
Twitter	111	40,205	6,364	4,568	771	1.92%	12.12%	649.59%
Total	1,799	663,384	36,866	29,819	4,658	0.70%	12.63%	258.92%

4. The size of the outbound data flow when the data is extracted based on the history. (See Section 6.2.2)

5. The size of outbound data flow when the repeated data flows are ignored.

An example of data extraction in Twitter's status update request is illustrated in Figure 6.2.

As shown in Table 6.2, the sizes of the extracted data can be reduced to an average of 12.63% of the data extracted by the simple protocol analysis. The extracted size is 0.7% of the total size of the HTTP request messages (Column 2 in the table). When compared with the total size of the user user

input (Column 1), extracted data is about 258.92% larger than the size of the actual data. This is signficantly reduced from the total HTTP request size (663,384/1,799 = 36875%) or the simple protocol analysis size (36,866/1,799 = 2049%).

Observations on the Cross-Domain Flows

The initial design decision about the cross-domain flow detection was made under an assumption that each Web application is built on a single Web server entry point even if the service is load-balanced at the back-end. However, while testing WebDLP, we encountered many example applications that do not satisfy this assumption.

Obviously, some of the Web applications consists of multiple Web servers interacting with the Web browsers, but which are not results of mashups, but due to the distributed architecture of the Web application. For example, when a Twitter user's Web browser accesses `http://twitter.com/`, some resources such as JavaScript files and images are downloaded from other servers such as `http://a0.twimg.com/twitter.js`, where `a0` can be different for each access. Although this is a legitimate part of the Web application, it will be detected as a cross-domain data flow when a string literal in `twitter.js` is sent to `http://twitter.com/`, because they belong to different domains. Similarly, Hotmail uses several different servers that do not even belong to the same domain.

Although the server names may change dynamically, there are some clear patterns in the servers that make up each Web application. The data extraction should be more effective if we treat each set of related servers as

belonging to the same domain.

6.4.2 Accuracy of the Similarity-based Classifier

We evaluated the accuracy of the similarity-based classifier in two ways. First, the stand-alone accuracy was measured by classifying the text data extracted from the test document set. Second, we built a simulated environment in which the classifier is integrated with the HTTP Monitor, the Data History Manager, and the Data Reposoitory to evaluate the overall accuracy of the integrated DLP solution. [1]

Accuracy as a Standalone Classifier

In order to determine the accuracy of the similarity-based classifier itself, we use three sets of office documents from three independent real-life projects, and then performed 10-fold cross-validation test with them. [2]

In the setup phase of the each round of test, the text data extracted from the documents in the test data sets are registered with the document repository along with the project name as a document class. In the test phase, we randomly chose some parts of the content from the test data sets from the three projects, and classify the content by using the similarity-based classifier. The effectiveness of the proposed method was evaluated by the *precision* and *recall*. The precision is defined as $TP/(TP + FP)$, while

[1] Note that the prototype supports end-to-end behavior from the HTTP monitor to classification, but the following evaluation was done in the simulated enviornment, because some real-life Web applications do not allow automatic posting of many messages and try to verify a human presence using CAPTCHA.

[2] In 10-fold cross validation, the data set from each project was split into 10 sets. In each round, 9 sets are used as the *training data sets* and the other set is used as the *test data set*. The test was repeated 10 times with a different test data set each test.

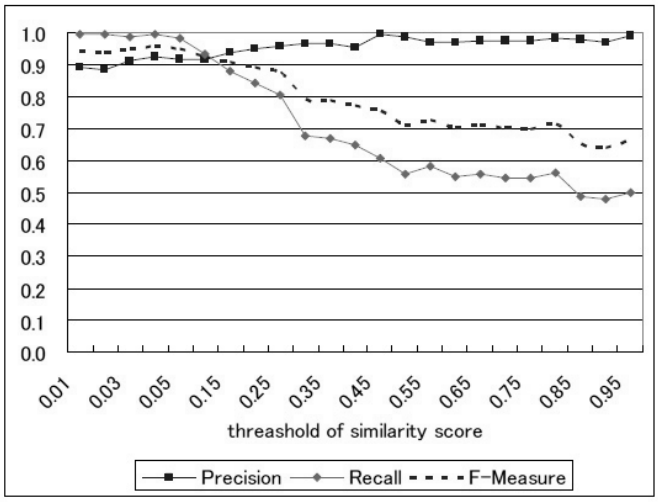


Figure 6.3: Classifier Accuracy: Standalone

the recall was defined as $TP/(TP+FN)$, where TP, FP, and FN represents true-positive, false-positive, and false-negative, respectively. The F-measure was defined as $(2 * precision * recall)/(precision + recall)$. Intuitively, given a document class, the precision represents the ratio of the correctly classified pieces of data among the data classified in that given class, and the recall represents how many pieces of the data that actually belong to the class were classified as that class. The F-measure is the harmonic-mean of the precision and the recall, taking both metrics into account.

The cross-validation test was repeated by changing the threashold of the similarity score, and the results are shown in Figure 6.3. The highest F-Measure, 0.9601 occurs when the threashold is 0.04, for which case the precision is 0.9264 and the recall is 0.9964. Based on these results, the accuracy of the similarity-based classifier is quite good in the stand-alone environment.

Accuracy of the Integrated WebDLP

In the second test we simulated the integrated DLP capabilities by combining the similarity-based classifier with the HTTP Monitor and the Data History Manager, to evaluate the accuracy of the classifier over all of the extracted data. We used the same 10-fold cross validation method as in Section 6.4.2 and the same Web applications as in Section 6.4.1. In each test, text data was randomly extracted from documents in test data sets and posted to the Web application as user input message.

The result of the test (Figure 6.4) showed that there are quite a lot of false positive matches when classifying data elements in the cross-domain and new data flows. The false positives are mainly caused by short pieces of data, because when using the TF-IDF algorithm, the likelihood of false positives increases when the data is short. For example, when the data being sent in the outbound flow is the name of a person, then any document which includes the same name will result in a high similarity score. More prevalent examples can be found in presentation slides with words such as "backup" or "Thank you, Any questions?". As a result, lots of short bits of data generated by the client-side JavaScript code coincidentally match the data in the document repository, causing false positives.

We modified the algorithm to ignore the short pieces of data that is less than 32 characters, because our scenario focus on data leakage from document reuse, and it is unlikely that leakage from document reuse occur with such short strings. The improved results are shown in Figure 6.5. The highest F-Measure of 0.8218 was obtained when the threashold was 0.1, when the precision was 0.7151 and the recall was 0.9660.

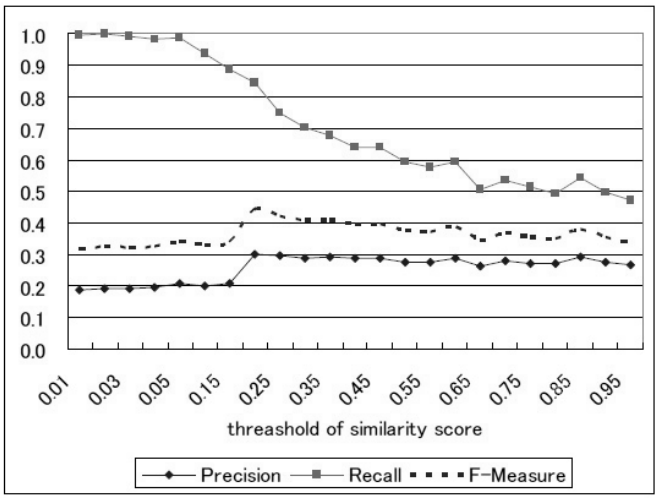


Figure 6.4: Classifier Accuracy: Simulated Web Application (including short strings)

Note that in both case, the recall itself does not suffer from the presence of the extra HTTP data, and thus the system can be tuned to prevent leakage with high probability at the cost of a higher false alarm rate. For example, by choosing a lower threashold such as 0.02, almost 100% of the data leakage can be prevented even though it generates about twice more false positives than true positives.

6.5 Discussion

This chapter proposed WebDLP, a proxy-based system to detect and prevent data leakage via Web browsers. We implemented a prototype system and demonstrated its effectiveness through experiments with three popular applications for Web-based e-mail and micro-blogging.

However, the proposed system still has some gaps that need be filled

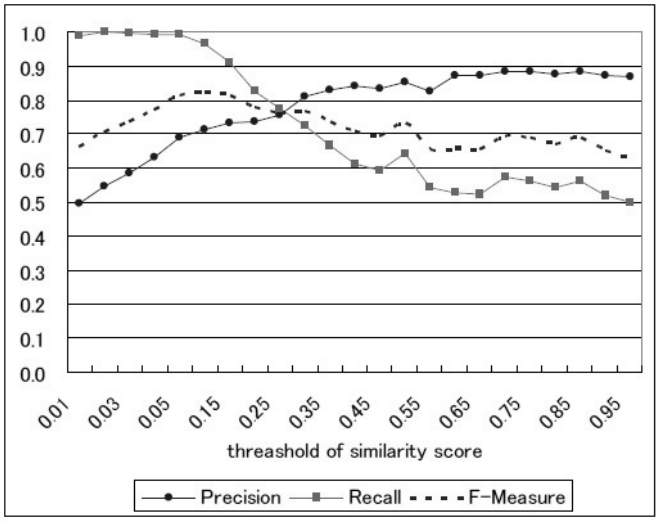


Figure 6.5: Classifier Accuracy: Simulated Web Application (ignoring strings shorter than 32 chars)

before being used in the real-world environments.

- When SSL is used to secure the client-server communication, the Web-DLP can still perform content analysis when it is configured as a man-in-the-middle for SSL connections. WebScarab has an MITM mode and thus the prototype WebDLP can analyze contents transferred over SSL. However, it causes security warnings in the Web browsers due to the mismatch of the server-side certificate and the URL being accessed, and some content is not properly downloaded. This is a common problem for all of the network-based DLP.

- When a user manually encrypts a message and sends it as, for example, a message body in a Web-based mail system, the proposed classifier cannot detect the class of the content. However, the content analysis can still detect that some new data flow is occurring, and assess the

amount of the information being leaked.

- The current model cannot detect covert channels by using metainformation, such as the number of data elements transmitted, the timing, or the number of non-confidential elements in an outbound data flow.

- The performance and memory consumption of the proposed system is out of the scope of this work. However, the implementation will need to be optimized for these aspects.

Our future research agenda include adapting some learning algorithms to learn the patterns of the data transmissions, and to ignore low-risk transmissions even if the data being sent is not identical to the past transmissions. In addition, the machine learning approach should be extended to detect covert channels.

In addition, the development of improved similarity detection algorithms that address DLP-specific requirements is an important challenge.

Chapter 7

Conclusion

This chapter is organized as follows. Section 7.1 summarizes the contributions made by this thesis. Section 7.2 outlines some of the future research directions.

7.1 Thesis Contributions

This thesis proposes three technologies to secure Web application environments by using information-flow control technologies.

Secure Browser Model

This first proposal is a novel Web browser security model that is built on the information-flow-based access control in the browser environment. The proposed model overcomes the dynamic nature of JavaScript to allow precise access control based on the origin of the code. The policy model addresses various threats to the Web applications that we are suffering today.

Although many proposals have been made recently to improve the secu-

rity design of Web browsers, most of the other proposals focus on improving the browser implementation, such as isolation of internal components in the Web browser to prevent interference between components. The overall security model in those proposals is the Same-Origin Policy, which is the de facto standard of the browser security model today. None of them enables fine-grained access control, especially based on the origin of the data that is tracked through program execution, as proposed in this thesis.

Most of the existing proposals for the mashup security are based on the idea of isolating each piece of content within its own sandbox, which is achieved either by using iframes in the current browsers or by extending the browser architecture. Although such approaches have the advantage of simplicity, there are drawbacks such as the loss of granularity and a less integrated user experience.

Content-Security Policy (CSP) is a new proposal by Mozilla [5] to prevent XSS attacks, which is the most relevant technology to the secure browser architecture presented in Chapter 4. CSP is a powerful solution to minimize the risks of XSS. On the other hand, it has a strong restriction that prevents existing Web applications from functioning properly. Although it is technically possible, it would be realistically difficult to convert existing Web applications to conform to CSP.

The secure browser model (Chapter 4) proposed in this thesis addresses these problem by fine-grained DOM-level access control, which allows tight integration of multiple services without isolating them in individual sandboxes. In addition, the proposed model can prevent CSRF and local network attacks by malicious services that happens to be part of a trusted mashup

application, by controlling its capability to access the network.

Dynamic Information-Flow Control Architecture (DIFCA)

DIFCA (Chapter 5) enforces the language-based information flow control policies for Java applications. DIFCA tracks the propagation of information in the program through most of the JVM instructions, and controls the input from and output to the external environment based on the given information flow policies. DIFCA also intercepts the JDBC queries to effectively label the query results, and control input to and output from the database.

DIFCA uses a bytecode rewriting technique to insert inline reference monitors (IRMs) into the application code, and thus has these advantages:

1. The IRMs can utilize the detailed conditions of the running applications, such as Java reflection.

2. It does not require the source code of the target application.

3. The system is independent of any JVM implementation.

DIFCA was influenced by [46] [47] [35], but the major difference is in the granularity of the label propagation. DIFCA supports information propagation through most of the JVM instructions, including arithmetic operations, array elements, multi-threading, and exceptions, and a policy based declassification mechanism. The work also integrated an intercepter for database queries, to effectively label database query results and to control input from and output to the database through JDBC.

Web-based Data Leakage Prevention (WebDLP)

The WebDLP (Chapter 6) technology consists of the protocol and the content analysis to identify sensitive data flows across domains (such as data flows from the on-premise domain to an off-premise domain, or between multiple off-premise domains), detection of the sensitivity of the data, and prevention of undesirable data flows. The proposed technology addresses the data flow risks when the modification of the client-side or the server-side runtime environment is not possible or realistic.

The contributions of the WebDLP include:

First, the WebDLP analyzes the massive Web traffic to extract the most interesting portions of the data flows. By recording the history of the past traffic and comparing the current data flows with the history, it can effectively extract cross-domain flows and user input data from outbound data flows.

Second, the WebDLP classifies the data based on the similarity to a known data set. The similarity-based classifier is designed based on an assumption that knowledge workers tend to frequently reuse and exchange documents a lot, by making new versions through group editing. The similarity-based classifier in the WebDLP checks two aspects: (1) whether the data in question is similar to specific documents with a known class and thus it should be classified as the same class, and (2) whether the data in question is common and prevalent among other classes and thus should not be classified as the specific class. By using the similarity-based classifier, the WebDLP can detect sensitive data that does not fall into specific patterns such as Personally Identifiable Information (PII).

The experiments with the prototype implementation showed the effec-

tiveness of the proposed methods in three real-life Web applications.

7.2 Future Work

Secure Browser Model

The current secure browser model omitted support for implicit flows because it has little effect under assumption of no conspiracy between trusted and untrusted content. Support of implicit flows will require further research due to the dynamic nature of JavaScript and the need for effective un-tainting or re-labeling. In this paper, various details of the browser behavior (e.g., SSL) have been omitted to keep the model simple. In addition, support for privilege overriding in a trusted code (similar to `doPrivileged` in Java) will also be needed in real-life applications. An effective and performable implementation is another topic that requires further research.

In addition, a drawback of the proposed secure browser model is its relative complexity in policy authoring. That is, the administrator of a Web application has to fully understand the data flows within the client-side JavaScript code precisely in order to write appropriate access control policies. On the other hand, CSP has its advantage in simple policy design, although it makes the client-side JavaScript code backward-incompatible. The future of the secure browser model can be envisioned as convergence of two approaches, seeking to find the right balance between the simplicity and the backward-compatibility.

Dynamic Information-Flow Control Architecture (DIFCA)

The current proposal DIFCA still leaves gaps for future research.

First, the purely dynamic approach can discover only information flows that are actually executed, and in particular cannot detect all of the implicit flows.

Second, the current approach requires a bytecode-level IRM to be inserted for every bytecode instruction of the original code, and this causes significant performance overhead.

Third, since DIFCA's internal table stores object references with associated security labels, it prevents target objects from being garbage collected, and causes memory overhead at run-time.

Fourth, terminating the transaction due to the information flow violation may cause problems in database consistency. It is inherently difficult to handle such exceptions without modifying the applications. Some of the problems may be acceptable when using DIFCA for the pre-deployment test, but care needs to be taken to define test cases with good coverage. A more realistic approach is to use DIFCA for just recording the log messages when violations occur, for the post-run auditing.

Usability and policy specification are also challenges that need to be addressed. DIFCA does not require the source code of the target applications, but the policy writer still needs to understand the structure of the program and the semantics of the methods. In particular, when a declassification policy is defined for a method, such a definition may easily introduce human error, unless the semantics of the method are well defined and the consequences of the declassification are well understood. This is a future topic for

allowing easy and safe policy definitions without needing knowledge of the source code.

Web-based Data Leakage Prevention (WebDLP)

When using the WebDLP as a proxy on an encrypted SSL channel, some public key certification issues have to be addressed to allow it work as a man-in-the-middle proxy without showing security alert messages on the Web browser.

In addition, when a user manually encrypts a message and sends it as, for example, a message body in a Web-based mail system, the proposed classifier cannot detect the class of the content. However, the content analysis can still detect that some new data flow is occurring, and assess the amount of information being leaked.

The current model cannot detect covert channels by using meta information, such as the number of data elements transmitted, the timing, or the number of non-confidential elements in an out-bound data flow.

The future research agenda include adoption of some learning algorithms to learn the patterns of data transmissions, and to ignore low-risk transmissions even if the data being sent is different. In addition, the machine learning approach should be extended to detect covert channels.

List of Publications

Full Papers in Refereed Journals

- Sachiko Yoshihama, Takaaki Tateishi, Naoshi Tabuchi, and Tsutomu Matsumoto, Information-flow-based Access Control for Web Browsers, IEICE Transactions Vol. E92-D, No. 5, pp. 836-850, The Institute of Electronics, Information and Communication Engineers, May 2009.

- Sachiko Yoshihama, Michiharu Kudoh, and Kazuko Oyanagi, Language-Based Information Flow Control in Dynamic Approach IPSJ Journal, Vol. 48, No. 9, pp. 3060-3072, Information Processing Society of Japan, September 2007. 吉濱佐知子, 工藤道治, 小柳和子, 「動的アプローチによる言語ベースの情報フロー制御」,『情報処理学会論文誌』 Vol.48 No.9 pp.3060-3072, 情報処理学会, 2007 年 9 月.

Papers in Refereed Conference Proceedings

- Sachiko Yoshihama, Takeo Yoshizawa, Yuji Watanabe, Michiharu Kudoh, and Kazuko Oyanagi, Dynamic Information Flow Control Architecture for Web Applications, in Proceedings of the 12th European

Symposium Research Computer Security (ESORICS 2007), Dresden, Germany, September 2007, Lecture Notes in Computer Science, Volume 4734/2008, pp. 267-282, Springer.

- Sachiko Yoshihama, Naohiko Uramoto, Satoshi Makino, Ai Ishida, Shinya Kawanaka, and Frederik De Keukelaere, Security Model for the Client-Side Web Application Environments, in Web 2.0 Security and Privacy (W2SP2007), May 2007.

Other Papers

- Sachiko Yoshihama, Ai Ishida, and Naohiko Uramoto, Typical Web 2.0 Attack Vectors and Countermeasures, IPSJ Magazine, Vol. 50, No. 1, pp. 44-54, Information Processing Society of Japan, January 2009. 吉濱佐知子， 石田愛， 浦本直彦， 「Web 2.0 アプリケーションにおける代表的な攻撃手法とその対策」 会誌『情報処理』 Vol. 50, No.1, pp. 44-54 情報処理学会 2009 年 1 月.

- Sachiko Yoshihama and Tsutomu Matsumoto, A Secure Browser Model for Web 2.0. Symposium on Cryptography and Information Security (SCIS 2008), January 2008. 吉濱佐知子， 松本勉， 「Web 2.0 のための安全なブラウザモデル」 暗号と情報セキュリティシンポジウム 2008 (SCIS 2008)，2008 年 1 月.

- Sachiko Yoshihama, Michiharu Kudoh, and Kazuko Oyanagi, Infora-tion Flow Control for Java with Inline Reference Monitors, Computer Security Symposium 2006 (CSS 2006), October 2006. 吉濱佐知子， 工藤道治， 小柳和子、「Inline Reference Monitor による Java 情報フロー

制御」 コンピュータセキュリティシンポジウム (CSS 2006), 2006 年 10 月.

Bibliography

[1] Access Control for Cross-Site Requests. http://www.w3.org/TR/access-control/.

[2] ADSafe. http://www.adsafe.org/.

[3] Apache Byte Code Engineering Library (BCEL). http://jakarta.apache.org/bcel/.

[4] Application Privacy Monitoring for JDBC (APM4JDBC). IBM AlphaWorks.

[5] Content Security Policy. http://people.mozilla.org/~bsterne/content-security-policy/index.html.

[6] Google Docs. http://docs.google.com/.

[7] HTML5 - a vocabulary and associated apis for HTML and XHTML. http://www.w3.org/html/wg/html5/.

[8] Linux intrusion detection system (lids). http://www.lids.org/.

[9] McAfee Data Loss Prevention. http://www.mcafee.com/japan/products/dlp.asp.

[10] RSA DLP. http://www.rsa.com/node.aspx?id=3426.

[11] Signed Scripts in Mozilla. http://www.mozilla.org/projects/security/components/signed-scripts.html.

[12] Technical explanation of the myspace worm.

[13] Tomoyo Linux. http://sourceforge.jp/projects/tomoyo/.

[14] XMLHttpRequest Level 2. http://www.w3.org/TR/XMLHttpRequest2/.

[15] Proposed ECMAScript 4th Edition - Language Overview. http://www.ecmascript.org/, October 2007.

[16] OpenAjax Alliance. OpenAjax Hub. http://sourceforge.net/projects/openajaxallianc.

[17] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *ACM CCS'08*, Alexandria, Virginia, USA, October 2008.

[18] Adam Barth, Collin Jackson, Charles Reis, and The Google Chrome Team. The security architecture of chromium. Technical report, Stanford University, Washington University & Google, Inc., August 2008.

[19] G. Barthe, A. Basu, and T. Rezk. Security types preserving compilation. In G. Levi and B. Steffen, editors, *Verification, Model-Checking and Abstract Interpretation (VMCAI04)*, 2004.

[20] G. Barthe, D. A. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *IEEE Symposyum on Security and Privacy*, 2006.

[21] Gilles Barthe and Tamara Rezk. Non-interference for a JVM-like language. In *Proceedings of the 2005 ACM SIGPLAN international workshop on Types in Languages Design and Implementation (TLDI 2005)*, 2005.

[22] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical report, MITRE Technical Report 2547, Volume I, 1973.

[23] Y. Beres and C.I. Dalton. Dynamic label binding at run-time. In *New Security Paradigms Workshop*, 2003.

[24] K.J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, Mitre, June 1977.

[25] Kevin Borders and Atul Prakash. Quantifying information leaks in outbound web traffic. In *IEEE Symposium on Security & Privacy*, 2009.

[26] Brian Chess, Yekaterina Tsipenyuk O'Neil, and Jacob West. Javascript hijacking. Technical report, Fortify Software, 2007.

[27] Cisco. Ironport.

[28] Doug Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627: `http://www.ietf.org/rfc/rfc4627.txt`, July 2006.

[29] Douglas Crockford. The module tag. `http://www.json.org/module.html`, October 2006.

[30] Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19 (5):236–243, 1976.

[31] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[32] Brendan Eich. ECMAScript Harmony. `https://mail.mozilla.org/pipermail/es-discuss/2008-August/003400.html`, 2008.

[33] Ulfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[34] The Apache Software Foundation. Apache Lucene. `http://lucene.apache.org/`.

[35] Michael Franz. Moving trust out of application programs: A software architecture based on multi-level security virtual machines. Technical Report No. 06-10, TR. 06-10, Donald Bren School of Information and Computer Science, University of California, Irvine, August 2006.

[36] Yusuke Furuta. "boku hamachi-chan konnichiwa konnichiwa!". `http://ascii.jp/elem/000/000/063/63560/`, September 2007.

[37] Samir Genaim and Fausto Spoto. Information flow analysis for Java bytecode. In *Verification, Model Checking and Abstract Interpretation (VMCAI05)*, volume LNCS 3385 of *Lecture Note in Computer Science*, pages 346–362, Paris, France, January 2005. Springer.

[38] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, 1997.

[39] Larry Greenemeier. New Attack Uses Bogus Web Sites To Deliver Malware . Information Week. URL: `http://www.informationweek.com/news/internet/showArticle.jhtml?articleID=201201582`, July 2007.

[40] Chris Grier, Shuo Tang, and Samuel King. Secure Web browsing with the OP Web browser. In *2008 IEEE Symposium on Security and Privacy*, May 2008.

[41] Jeremiah Grossman. Whitehat website security statistics report, October 2007.

[42] Jeremiah Grossman. Whitehat website security statistics report, March 2008.

[43] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David Schmidt. Automata-based confidentiality monitoring. In *Proceedings of the 11th Annual Asian Computing Science Conference (ASIAN'06)*, 2006.

[44] Carolyn Guevarra. Another malware pulls an Italian job. TrendLabs Malware Blog. URL: `http://blog.trendmicro.com/another-malware-pulls-an-italian-job/`, June 2007.

[45] ha.ckers.org. Xss (cross site scripting) cheat sheet esp: for filter evasion. `http://ha.ckers.org/xss.html`.

[46] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for Java. In *Annual Computer Security Applications Conference (ACSAC)*, 2005.

[47] Vivek Haldar, Deepak Chandra, and Michael Franz. Practical, dynamic information flow for virtual machines. In *the 2nd International Workshop on Programming Language Interference and Dependence, September 2005 (PLID 2005)*, 2005.

[48] Oystein Hallaraker and Giovanni Vigna. Detecting malicious JavaScript code in Mozilla. In *Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, 2005.

[49] Robert Hansen and Jeremiah Grossman. Clickjacking. `http://www.sectheory.com/clickjacking.htm`, 2008.

[50] Boniface Hicks, Kiyan Ahmadizadeh, and Patrick McDaniel. From languages to systems: Understanding practical application development in security-typed languages. In *Proceedings of the Annual Computer Security Applications Conference 2006 (ACSAC 2006)*. Systems and Internet Infrastructure Security Laboratory (SIIS) Computer Science and Engineering, Pennsylvania State University, December 2006.

[51] Billy Hoffman. JavaScript Malware for a Gray Goo Tomorrow! http://h71028.www7.hp.com/enterprise/downloads/Javascript_malware.pdf, 2008.

[52] Jon Howell, Collin Jackson, Helen J. Wang, and Xiaofeng Fan. Mashupos - operating system abstractions for client mashups. In *11th Workshop on Hot Topics in Operating Systems (HotOS XI), San Diego, CA, May 7-9, 2007*, 2007.

[53] IBM. Projectzero. http://www.projectzero.org/.

[54] Bob Ippolito. Remote JSON - JSONP. http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/, December 2005.

[55] Omar Ismail, Masashi Eto, and Youki Kadobayashi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability'. In *Proceedings of the 18th International Conference on AdvancedI nformation Networking and Applications (AINA2004), March 2004*, 2004.

[56] Collin Jackson and Adam Barth. Beware of finer-grained origins. In *Web 2.0 Security and Privacy 2008 (W2SP 2008)*, 2008.

[57] Collin Jackson and Helen Wang. Subspace: Secure cross-domain communication for web mashups. In *16th International World Wide Web Conference May 8-12, 2007, Banff, Alberta, Canada*, May 2007.

[58] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser enforced embedded policies. In *Proceedings of*

the 16th international conference on World Wide Web (WWW 2007), 2007.

[59] Martin Johns and Justus Winter. Requestrodeo: Client side protection against session riding. In *OWASP Europe Conference, May 2006*, 2006.

[60] Nenad Jovanovic, Engin Kirda, and Christopher Kruegel. Preventing cross site request forgery attacks. In *IEEE International Conference on Security and Privacy in Communication Networks (SecureComm)*, August 2006.

[61] Frederik De Keukelaere, Sumeer Bhola, Michael Steiner, Suresh Chari, and Sachiko Yoshihama. Smash: Secure component model for cross-domain mashups on unmodified browsers. In *Proceeding of the 17th international conference on World Wide Web (WWW 2008)*, Beijing, China, April 2008.

[62] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *Proceedings of the 2006 ACM symposium on Applied computing, Dijon, France, 2006.*, 2006.

[63] Amit Klein. DOM based cross site scripting or XSS of the third kind. http://www.webappsec.org/projects/articles/071105.shtml, July 2005.

[64] Naoki Kobayashi and Keita Shirane. Type-based information flow analysis for low-level languages. In *APLAS 2002*, pages 2–21, 2002.

[65] Leonard J. LaPadula and D. Elliott Bell. Secure computer systems: A mathematical model. Technical report, MITRE Technical Report 2547, Volume II, 1973.

[66] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 158–170, New York, NY, USA, 2005. ACM Press.

[67] Peng Li and Steve Zdancewic. Practical information-flow control in Web-based information systems. In *Proceedings the 18th IEEE Computer Security Foundation Workshop (CSFW)*, June 2005.

[68] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[69] Benjamin Livshits and Ulfar Erlingsson. Using Web application construction frameworks to protect against code injection attacks. In *Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, San Diego, California, June 2007.

[70] Peter A. Loscocco and Stephen D. Smalley. Meeting critical security objectives with Security-Enhanced Linux. In *Ottawa Linux Symposium*, 2001.

[71] Ziqing Mao, Ninghui Li, and Ian Molloy. Defeating cross-site request forgery attacks with browser-enforced authenticity protection. In *Financial Cryptography and Data Security (FC'2009)*, volume Lecture

Notes in Computer Science Volume 5628/2009, Accra Beach, Barbados, February 2009.

[72] Robert McMillan. New URI browser flaws worse than first thought. IDG News Service. URL: http://security.itworld.com/5043/070815URIbrowserflaw/page_1.html, August 2007.

[73] Microsoft. Better AJAX development: Windows internet explorer 8. Whitepaper, March 2008.

[74] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Caja - safe active content in sanitized JavaScript. Technical report, Google, January 2008.

[75] Rich Mogull. Understanding and selecting a data loss prevention solution. Technical report, Securosis, L.L.C.

[76] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[77] Andrew C. Myers and Andrei Sabelfeld. Enforcing robust declassification. In *proceedings of CSFW 2004*, 2004.

[78] OWASP. AntiSamy project. http://www.owasp.org/index.php/Category:OWASP_AntiSamy_Project.

[79] The Open Web Application Security Project (OWASP). OWASP WebScarab Project. http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project.

[80] Marco Pistoia, Anindya Banerjee, and David A. Naumann. Beyond stack inspection: A unified access-control and information-flow security model. In *IEEE Symposium on Security & Privacy*, pages 149–163, May 2007.

[81] Gordon D. Plotkin. A structural approach to operational semantics. Tech Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark, 1981.

[82] Naizhen Qi. *Efficient and scalable access control enforcement of XML documents*. PhD thesis, Graduate School of Information Science and Engineering, Tokyo Institute of Technology, Japan, 2007.

[83] Jesse Ruderman. The same origin policy. `http://www.mozilla.org/projects/security/components/same-origin.html`, August 2001.

[84] Andrei Sabelfeld and Andrew C. Myers. Language-based information flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[85] Paritosh Shroff, Scott F. Smith, and Mark Thober. Dynamic dependency monitoring to secure information flow. In *the 20th IEEE Computer Security Foundations Symposium*, 2007.

[86] Sid Stamm, Zulfikar Ramzan, and Markus Jackobsson. Drive-by pharming. Technical Report TR641, Indiana University, December 2006.

[87] Palisade Systems. Packetsure. `http://www.palisadesys.com/products/packetsure/`.

[88] Takaaki Tateishi and Naoshi Tabuchi. Secure behavior of Web browsers to prevent information leakages. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC '07)*, pages 65–72, Washington, DC, USA, 2007. IEEE Computer Society.

[89] US-CERT. Vulnerability Note VU 808921: eBay contains a cross-site scripting vulnerability. `http://www.kb.cert.org/vuls/id/808921`, April 2006.

[90] Sven Vetsch. Xsio - cross site image overlaying, August 2007.

[91] Philipp Vogt, Florian Nentwich, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *14th Annual Network and Distributed System Security Symposium (NDSS 2007), San Diego, CA, February 2007.*, February 2007.

[92] Common Vulnerabilities and Exposures. CVE-2005-4089 cross-domain information disclosure vulnerability. `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-4089`, 2005.

[93] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the Gazelle Web browser. In *Usenix Security 2009*, pages 417–432, 2009.

[94] Aaron Weaver. Cross site printing. November 2007.

[95] Wikipedia. TF-IDF. `http://en.wikipedia.org/wiki/Tf-idf`.

[96] Sachiko Yoshihama, Ai Ishida, and Naohiko Uramoto. Typical Web 2.0 attack vectors and countermeasures. In *IPSJ Magazine*, volume 50, pages 44–54, January 2009.

[97] Sachiko Yoshihama, Michiharu Kudoh, and Kazuko Oyanagi. Infora-tion flow control for Java with inline reference monitors. In *Computer Security Symposium (CSS2006)*, October 2006.

[98] Sachiko Yoshihama, Michiharu Kudoh, and Kazuko Oyanagi. Language-based information flow control in dynamic approach. *IPSJ Journal*, 48(9):3060–3072, September 2007.

[99] Sachiko Yoshihama and Tsutomu Matsumoto. A secure browser model for Web 2.0. In *Symposium on Cryptography and Information Security (SCIS 2008)*, January 2008.

[100] Sachiko Yoshihama, Takaaki Tateishi, Naoshi Tabuchi, and Tsutomu Matsumoto. Information-flow-based access control for web browsers. *IEICE Transactions*, E92-D(5), May 2009.

[101] Sachiko Yoshihama, Takeo Yoshizawa, Yuji Watanabe, Michiharu Kudoh, and Kazuko Oyanagi. Dynamic information flow control architecture for Web applications. In *the 12th European Symposium Research Computer Security (ESORICS 2007)*, Dresden, Germany, September 24-26 2007.

[102] Dachuan Yu and Nayeem Islam. A typed assembly language for confidentiality. In *2006 European Symposium on Programming (ESOP'06), Vienna, Austria, March 2006. LNCS Vol. 3924.*, 2006.

[103] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Symposium on Operating Systems Principles, 2001 (SOSP 2001)*, 2001.

[104] Lantian Zheng, Stephen Chong, Andrew C. Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *IEEE Symposium on Security and Privacy*, 2003.