NORTHWESTERN UNIVERSITY

Protecting Client Browsers with a Principal-based Approach

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

DOCTOR OF PHILOSOPHY

Field of Computer Science

By

Yinzhi Cao

EVANSTON, ILLINOIS

June 2014

ABSTRACT

Protecting Client Browsers with a Principal-based Approach

Yinzhi Cao

Web-based attacks have become very prevalent on the Internet. As conducted by Kaspersky lab, 34.7% of their users were targeted by a web-based attack at least once while browsing the Internet. In my thesis, I will discuss three fundamental tasks in building a principal-based browser architecture preventing various web attacks. In the first task, I will introduce how to prevent and detect attacks, such as drive-by download, which penetrate the boundary of a browser principal. In particular, I will present JShield, a vulnerability-based detection engine that is more robust to obfuscated drive-by download attacks, when compared to various anti-virus software and most recent research papers. Then, in the second task, I will introduce configurable origin policy (COP), the access control policy of next generation browsers, which gives websites more freedom to render various contents inside one principal. In the end, I will also introduce the communication between different browser principals.

Acknowledgments

I would like to express the deepest appreciation to my committee chair and advisor, Professor Yan Chen, who supports me through my PhD life and guides me into the ocean of computer security and privacy. Without his help this dissertation would not have been possible.

I would like to thank my committee members, Professor V.N. (Venkat) Venkatakrishnan from University of Illinois Chicago and Professor Aleksandar Kuzmanovic from Northwestern University, who have provided numerous opinions during my thesis proposal. Professor V.N. (Venkat) Venkatakrishnan also gives me a wealth of comments for my Virtual Browser work.

I would also thank Doctor Alexander Moshchuk (another committee member of mine) from Microsoft Research and Doctor Zhichun Li from NEC Labs, who co-authored the paper related to configurable origin policy and contributed a lot to the idea and the final paper.

In addition, I would like to express a thank you to Professor Giovanni Vigna and Professor Christopher Kruegel, both of whom are my internship mentors at University of California Santa Barbara and have contributed to the last part of my thesis about the communication between different browser principals.

Meanwhile, I also want to thank Director Phillip Porras and Doctor Vinod Yegneswaran, my internship mentors at SRI International. They have contributed a lot to the second part of the thesis related to JavaScript worms.

Last but not least, I would like to thank Professor Bin Liu, Professor Guofei Gu, Professor Jianwei Zhuge, Professor Kai Chen, Professor Peter Dinda, Doctor Ming Zhang, Doctor Manuel

Egele, Doctor Shuo Chen, Antonio Bianchi, Chao Yang, Clint Sbisa, Gianluca Stringhini, Hongyu Gao, Kevin Borgolte, Vaibhav Rastogi, Xiang Pan, Xitao Wen, Yao Zhao, Yan Shoshitaishvili, Yanick Fratantonio, Zhengyang Qu, my fiancee, my parents, numerous anonymous reviewers of my papers, and you who are reading my thesis.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

As the advent of Web 2.0, more and more new types of attacks has evolved on the new platform. As shown in the Symantec Internet Threat Report, there is 93% increase of web attacks of the year 2010 [**44**]. One of those attacks, cross-site scripting, is the No. 1 of top ten vulnerabilities as reported by WhiteHat Web site Security Statistics Report [**48**]. The McAfee Q1 2011 report II also states that they found 57,000 URLs with browser exploits (drive-by download) for the quarter [**28**].

My thesis research aims to build a principal-based browser architecture to detect, quarantine, and prevent various Web-borne security threats, which happen within, across, and outside Web principals[1]. As an analogy, if we consider a Web principal as a bottle, and the resource as the liquid inside the bottle, the task of my research is to pour the liquid into its corresponding bottle, and then give the bottle the correct label, also known as *origin* in the context of Web security. To achieve the goal, there are three fundamental questions to answer.

(1) How to make and strengthen a principal? A principal defines the boundary of resources isolated from other principals, and the penetration of other principals will cause a privilege escalation attack. Therefore, researchers need to build a principal as strongly as possible. Considering the bottle and liquid analogy, we need to prevent liquid from leaking out of a bottle by strengthening the bottle.

[1]In this statement, a Web principal, as borrowed from operating systems and defined by existing work [**152**], is an isolated security container of resources inside the client browser.

Two pieces of my work, Virtual Browser [**75, 76**] and WebShield [**114**], fall into this category. Virtual Browser, a concept comparable to a virtual machine, strengthens a browser principal by creating a virtualized browser to sandbox third-party JavaScript, and prevent drive-by download and cross-site scripting (XSS) attacks. My other work, WebShield, moves heavy and potentially dangerous client-side computation to a proxy sitting in between a client and a Web server. In other words, WebShield strengthens a browser principal by reducing it to a thin terminal, and lets the proxy handle the vulnerable contents.

Like two sides of a coin, defense and detection are complementary to each other. My two other pieces of work, JShield [**77**] and MPScan [**118**], try to detect those malicious contents penetrating existing principals, a/k/a drive-by download attacks, in normal Web traffic and Adobe Portable Document File (PDF). In those two works, I propose an opcode level vulnerability signature, including a definitive state automaton, plus a variable pool, to represent the inherent property of a certain vulnerability, and use the signature to match the opcode sequence containing an exploit. Both systems have been adopted by Huawei, the world largest telecommunication equipment maker, and filed in a U.S. patent.

(2) What contents are to put inside a principal? A principal, being empty after creation, is filled by a server and/or other principals. A careless allocation of contents inside a principal will lead to an XSS attack, comparable to the case where a mixture of vinegar and soda causes the explosion of the bottle. After contents allocation, an immediate task is to label a principal, *i.e.*, giving each principal a name that can be distinguished from other principals. For example, a server can differentiate requests from various client-side principals by the attached *origin* header. Similarly, a bottle should have a label to show the contents inside, such as water, vodka, and a cocktail mixed from vodka and gin, so that people could choose the correct bottle that they want.

There are two types of allocation algorithms: allocation based on web application developer's choice and allocation based on a purely automatic algorithm. Two pieces of my work, Configurable origin policy (COP) [**79**] and PathCutter [**81**], fall into the former. Another piece of my work, TrackingFree, fall into the latter. Configurable origin policy (COP) [**79**] lets the principal itself manage the contents allocation. With a secret identity, each principal can easily merge or split from another principal. Correspondingly, PathCutter selects user-generated contents, i.e., those that may contain malicious JavaScript, and puts them inside an isolated Web principal. Although XSS attacks can still succeed, PathCutter cuts off the propagation path of the attacks, and mitigates further infections to other users. On contrary, TrackingFree automatically recognize contents at client browser and allocate them into different principals based on an indegree-bounded graph.

Meanwhile, I also have some measurement study on misplaced contents, *i.e.*, measurement of add-on XSS, a type of XSS initiated by a user's fault from the client side. The work focuses on how and when a user mistakenly puts malicious contents into a Web principal. We recruit Amazon Mechanical Turks, and study their behavior given an add-on XSS.

(3) How to connect two principals? Principals, being isolated in a Web browser, sometimes need to talk to each other through a well-established channel, e.g., in the case of single-sign on (SSO), a relying party such as *New York Times* needs to talk to an identity party such as Facebook for a user's credential. When coming back to the bottle analogy, the channel is similar to a pipe between two bottles to transfer liquid. My work on securing Web single sign-on [**80**] fall in the category, which creates a bi-directional, secure channel upon the existing *postMessage* channel between two principals.

The thesis is organized as follows. Chapter 2 states how to build a strong principal. Then, Chapter 3 discusses contents allocation policy. In the end, Chapter 4 presents the communication between different principals.

CHAPTER 2

Building Strong Principals

2.1. Principal Spectrum

Historically, when seeking the balance between security and performance, there are many methods of defining a browser principal as shown in Figure 2.1. Traditional browsers, such as IE and Firefox, adopt a fast thread-based approach in its principal. Then, to achieve better security, Google Chrome and OP Browser adopt a process-based approach to isolate different memory states. The isolation is strong, but still cannot prevent some of the drive-by download attacks. Therefore, I propose a even stronger principal - virtual browser, a virtualized browser running upon the native browser to further sandbox contents from different origins. Virtual browser has a strong isolation mechanism, however both native browser and virtual browser are running upon the same operating system. Some researchers [**85**] also propose virtual machine based isolation to implement a principal. In the case of virtual machine based principal implementation, there is only

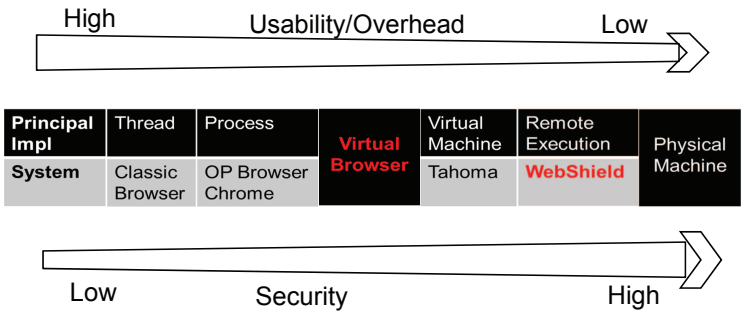

Figure 2.1. Principal Spectrum.

one physical machine for a browser. To further strengthen security, one of my work - WebShield adopts a proxy, which renders dynamic contents, and turns the client-side browser into a thin terminal. Obviously, one can also propose the strongest or imaginary browser that renders contents from different origins in different physical machines. However, such an implementation will make the browser unusable.

In this chapter, I will first introduce one of my work - Virtual Browser, which make a strong principal. Then, I will present another of my work - JShield, which detects malicious contents, which leak from one principal to the operating system.

2.2. A Strong Principal (Virtual Browser)

Modern web sites often use third party JavaScripts to enrich user experiences. Web mashups combine services from different parties to provide complex web applications. For example, a web site may use JavaScript games from other party to attract users, include JavaScript code from targeted advisement companies for increasing revenue, embed a third party counter to record the number of visited users, and enable third party widgets for richer functionalities. In each of these cases, some third party JavaScripts, which are not developed by the web site, have the same privileges as the JavaScript code from the web site itself. Although these third party JavaScripts enrich the functionalities of the web site, malicious third party Javascripts can potentially fully subvert the security policy of the web site, and launch all kinds of attacks.

Therefore, in this paper, we propose Virtual Browser, a virtualized browser built on top of a native browser to sandbox third-party JavaScript. The idea of Virtual Browser is comparable to that of virtual machines. It is written in a language that a native browser supports, such as JavaScript, so that no browser modification is required. Virtual Browser has its own HTML parser, CSS parser, and JavaScript interpreter, which are independent from the native browser. Third party JavaScripts

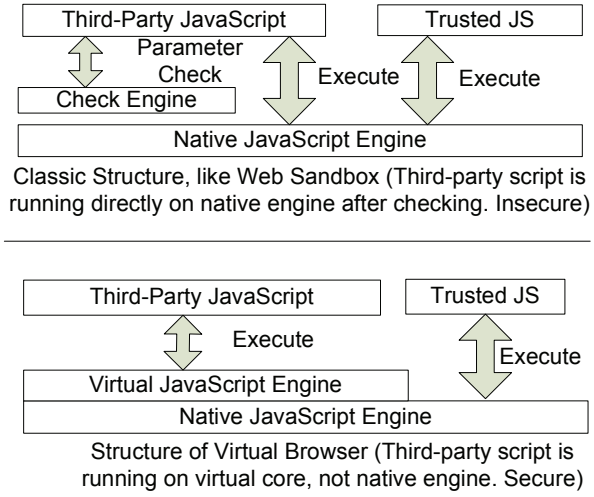

Figure 2.2. Classical Structure vs. Virtual Browser

are parsed only once in Virtual Browser and run on top of the virtual JavaScript interpreter. The untrusted third party JavaScripts are isolated from the trusted JavaScripts of the web site by design. Virtual Browser introduces only the necessary communications between the JavaScripts from the web site and the third party JavaScripts.

Existing works such as Microsoft Web Sandbox [**126**] and Google Caja [**94**] may also be thought of as employing virtualization but our technique is significantly different. The key difference is whether third party JavaScripts are directly running on a native JavaScript engine. Figure 2.2 illustrates the difference. Virtual Browser executes third party JavaScripts on a virtualized JavaScript engine; on the other hand, existing approaches check the parameters of each third party JavaScript expression and then let them execute directly on the native JavaScript engine. Web Sandbox[**126**] makes a big step toward virtualization. It provides a virtualized environment for native execution of third party JavaScripts, but its execution is still restricted by the parameter checking model. As shown in Figure 2.3(a), it provides a virtualized environment for *for* loop. All the variables are fetched from the virtualized environment. However, the *for* loop itself is still

running on a native JavaScript engine[1]. As shown in Section 3.1.1, this is the reason why they are vulnerable to unknown native JavaScript engine vulnerabilities and it is hard for them to handle dynamic JavaScript features like *eval* and *with*.

Security is the key property of our design. In order to make Virtual Browser secure, we need to prevent third party JavaScripts from directly running on a native JavaScript engine. Two methods are used here to achieve our design: *avoidance* and *redirection*. Avoidance means that we avoid using some dangerous functions in the native browser when implementing Virtual Browser. The dangerous functions are functions that potentially lead a JavaScript string to be parsed and executed on the native JavaScript engine. For example, native *eval* in JavaScript can execute a string. If *eval* is not used appropriately, third party scripts, which are input to Virtual Browser as a string, can flow to the native *eval* and get executed. Hence, we do not use the native *eval* function when implementing the virtual browser. This ensures that there is no way for third party JavaScripts to exploit the Virtual Browser to access the native *eval* function. Redirection means we redirect data flows to a place that is ensured to be secure. For example, third party JavaScripts may leak to the native JavaScript engine through calls to the function *setTimeout*. Instead of letting the flow go into the native JavaScript engine, we redirect it back to the virtual JavaScript engine. We will present the details about how we use these two methods in Section 3.1.4.

The execution speed of our system is similar to that of Web Sandbox [**126**] and is fast enough to sandbox reasonable lengths of third party JavaScripts as discussed in Section 2.2.5.1. We find that even animation scripts, which trigger events much more frequently than average scripts, can work well with Virtual Browser.

Contributions: We make the following contributions.

[1]Microsoft Web Sandbox actually transforms the for loop into $for(e(b,"i",0); c(b,"i") < 10; i(b,"i",1))$ for conciseness. Since the sentence is difficult to be read, we change its presentation to be a human-readable manner as shown in Figure 2.3.

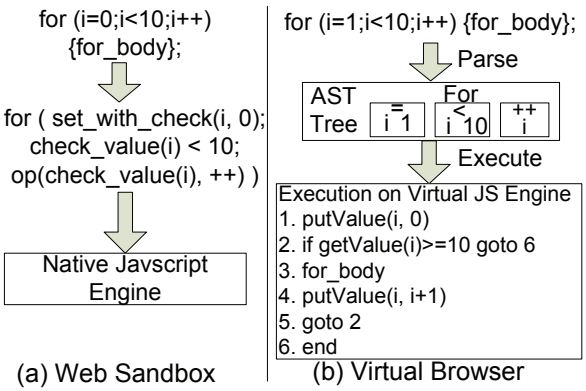

Figure 2.3. Comparison of Web Sandbox and Virtual Browser when Executing For-loop

- **Virtualization.** We propose the concept of Virtual Browser and adopt virtualization to provide security. Although earlier works may be conceived of as implementing some sort of virtualization, it is incomplete; we are the first to provide browser-level virtualization.
- **Enhanced Security for Isolation[2].** As shown in Section 3.1.1, compared to native sandboxes [**125, 148**] that purely rely on *iframe* for isolation, Virtual Browser is more robust to unknown native JavaScript engine vulnerabilities.
- **Securing even traditionally Unsafe Features in JavaScript.** A recent measurement study [**163**] reveals that 44.4% of their measured web sites use *eval*. In addition, Guarnieri et al. [**97**] shows that 9.4% of widgets use *with*. Thus, it is important to secure these functionalities.

2.2.1. Motivation and Related Work

Related work in securing third-party JavaScript can be classified into two categories: approaches using JavaScript features and approaches using native browser features.

[2]For securing communication between trusted and third party JavaScripts, Virtual Browser DOES NOT improve state-of-the-art.

Table 2.1. Comparing Virtual Browser with Existing Approaches

	Approaches with native browser support			JavaScript level approaches			
	Modifying browsers [**125, 56**]	Approaches Using NaCl[**91**]	Using *iframe* [**148, 86**]	Static methods [**2, 162, 121, 131**]	Mixed methods [**97, 101**]	Runtime approaches: Others [**94, 126**]	Runtime approaches: Virtual Browser
Robust to browser quirks	Yes	Yes	Yes	No	No	No	Yes
Robust to drive-by-downloads	No	Yes	No	No	Partial	Partial	Yes
(1) Caused by unknown JavaScript engine vulnerabilities	No	Yes	No	No	No	No	Yes
(2) Caused by others	No	Yes	No	No	Yes	Yes	Yes
Dynamic JavaScript feature support (like eval and with)	Yes	Yes	Yes	No	No	No	Yes
Support by all browsers	No	No	Yes	Yes	Yes	Yes	Yes
Speed	Fast	Fast	Fast	Fast	Medium	Slow	Slow

Securing Third-party JavaScript by Using JavaScript Features. Existing works about securing third-party JavaScript by using JavaScript features can be classified into three sub-categories: static, runtime, and mixed approaches.

- *Static Methods.* Many approaches, such as ADSafe [**2**], FBJS [**15**], CoreScript [**162**], and Maffeis et al. [**121**], restrict JavaScript to a subset and perform static check and enforcement upon third-party JavaScript.

- *Runtime Methods.* Microsoft Web Sandbox[**126**] and Google Caja[**94**] rewrite JavaScript to wrap up every dangerous part and perform runtime check with its own libraries. BrowserShield[**132**] and Kikuchi[**112**] are middle box solutions put at a proxy or a gateway. They wrap JavaScript with checking code at the middle box and perform runtime check at client side.

- *Mixed Methods.* Gatekeeper[**97**], a mostly static method also contains some runtime methods. It performs a points-to analysis on third-party JavaScript code and deploys policies during runtime. Huang et al. also check information flow at client side and protect client at runtime[**101**].

Securing Third-party JavaScript by Native Browser Features. Approaches using native browser features can be classified into three sub-categories: modifying native browser, using plugin, and using *iframes*.

- *Browser Modification.* ConScript [**125**]/WebJail [**56**] modify the IE8/Firefox browser kernel to enforce the deployment of policies (advices). MashupOS [**151**] proposes a new HTML tag *Sandbox* to secure Mashups by modifying native browsers. OMash [**86**] modifies Firefox to adopt object abstraction and isolate Mashups.
- *Using Plugins.* AdSentry [**91**] executes third-party codes in a shadow JavaScript engine sandboxed by Native Client [**161**].
- *Using iframes.* AdJail [**148**] and SMash [**89**] proposes using iframes to isolate third-party JavaScript.

Comparing with Virtual Browser. We discuss four important points: (*a*) Robustness to browser quirks, (*b*) Robustness to unknown native JavaScript engine vulnerabilities, (*c*) Support of some dynamic language features, and (*d*) Support by all existing browsers. The comparison is shown in Table 2.1.

First, we show how browser quirks, non-standard HTML and JavaScript, influence present approaches. All existing web browsers support browser quirks, because web-programmers are humans, and mistakes are very likely during programming. Browser quirks have previously been well studied in BluePrint[**149**] and DSI [**128**]. For example, the server side filter on the Facebook server had such vulnerability [**142**]. A string *<img src="..." onload:=attackcode>* is interpreted as *<img src="..." onload = attackcode>* at browsers (Firefox 2.0.0.2 or lower) but as *<img src="...">* at the server-side filter.

All existing JavaScript level approaches [**97, 2, 131, 126, 94**] define a particular server side (or middle-box) interpretation which may be very different from the browsers' interpretation, and

hence remain vulnerable to such attacks. In Virtual Browser, those attacks will not happen because scripts are parsed only once at client-side.

Second, existing native sandboxing approaches that purely rely on *iframe* for isolation, like AdJail [**148**] and SMash [**89**], and JavaScript level approaches, like Web Sandbox [**126**], are vulnerable to unknown native JavaScript engine vulnerabilities but Virtual Browser is more robust to unknown attacks.

We take the for loop as an example in Figure 2.3 again. Assume there is an unknown integer overflow in the for loop of native browser that is triggered by $for(i = a; i < b; i = i+c)$ when $a, b,$ and c are certain values. For a native sandbox, the vulnerability can be directly triggered. Because the vulnerability is unknown, neither Web Sandbox nor BrowserShield [**132**] (from which Web Sandbox evolved) can check the parameters and spot the attack. The vulnerability can still be directly triggered because as shown in Figure 2.3(a), the for loop is running on native JavaScript engine.

In Virtual Browser, since it interprets the for loop, direct input of the for loop will not trigger the vulnerability. As shown in Figure 2.3(b), the for loop is not running directly on the native JavaScript engine. In order to compromise the native browser with this vulnerability, virtual browser source code needs to have the sentence with exactly the same pattern: a for-loop where a, b and c are all open inputs. Then attackers need to manipulate other inputs to let the sentence in virtual browser source code get the certain values that can trigger the attack. Either of the two conditions is not easy to satisfy, as evaluated in Section 2.2.5.3.

Third, some dynamic language features, such as *eval* and *with*, are not supported in present JavaScript level approaches. Developers however still use *eval* to parse JSON strings in old browsers with no native JSON support. Gatekeeper [**97**] reveals that about 9.4% of widgets use *with*.

As shown in Figure 2.2, the classical runtime approaches (such as the runtime part in GateKeeper [**97**]) employ a parameter checking model, implying that they cannot check the safety of *eval* and *setTimeout*, whose parameters contain JavaScript code. and need to be passed to the JavaScript parser. Web Sandbox [**126**] itself does not execute third party scripts, and therefore it is hard to switch execution contexts for *with* statement. Meanwhile, although it is possible to recursively transfer arguments of *eval* back to the server for further transforming, large client-server delays will occur and render the approach extremely slow. Therefore, in the implementation of Web Sandbox, *with* is not supported and the support of *eval* is incomplete.

Fourth, those approaches, which modify existing browsers or utilize plugins like Native Client [**161**], are not supported by all existing browsers. Mozilla publicly rejects adopting NaCl [**30**], and meanwhile there is not clue that IE and Opera will adopt NaCl either. Therefore, those approaches can protect only a limited number of users who deploy their approaches. Virtual browser uses only JavaScript features that are supported by all present browsers.

2.2.2. Design

In Section 2.2.2.1, we first introduce the architecture of Virtual Browser. Then we give several JavaScript examples to show how exactly Virtual Browser works in Section 2.2.2.2.

2.2.2.1. Architecture. The architecture of Virtual Browser is shown in Figure 2.4. Virtual Browser is very similar to a native web browser except that it is written in JavaScript. We will introduce the interface, components and flows of Virtual Browser below.

Interface. Similar to Microsoft Web Sandbox, Virtual Browser takes a string, which contains the code of a third-party JavaScript program, as input. For example, we are using the following codes to include third-party JavaScripts.

```
<script> evaluate(str); </script>
```

str is a string that represents a third-party JavaScript code, which can be embedded inside host web pages. Maintenance of *str* is vulnerable to string injection attacks. In our approach, we leverage Base64 encoding, one of the many existing ways [**149, 128**] to prevent string injection attacks.

Virtual Browser also provides a file loading interface.

```
<script> run("http://www.a.com/JS/test.js"); </script>
```

An $XMLHTTPRequest$ will be made to the same origin web server (where all third-party and trusted codes and Virtual Browser are fetched) first by Virtual Browser. The same origin web server will redirect the request to the real web server (`www.a.com`). Therefore received contents will be fed into the aforementioned *evaluate* interface.

Components and Data Objects. The functionality of these components and data objects in Virtual Browser is similar to their corresponding parts in native browser.

Components of Virtual Browser include a virtual JavaScript parser, a virtual JavaScript execution engine, a virtual HTML parser, a virtual CSS parser and so on.

- *Virtual JavaScript Parser:* It parses third-party JavaScript codes and outputs the parsed JavaScript AST tree.
- *Virtual JavaScript Execution Engine:* It executes the parsed JavaScript AST tree from virtual JavaScript parser. The interface of the JavaScript execution engine has three parts: *putValue*, *getValue* and *function call/return*. *putValue* is loaded every time an object is changed. Every modification to a private (defined by third-party) function/variable or a shared (from trusted scripts or third-party) function/variables goes through *putValue*. *GetValue* provides an interface for every read operation. *Function call/return* are used for calling shared functions from natively running code and private functions from third-party codes. Our design of the interface of the

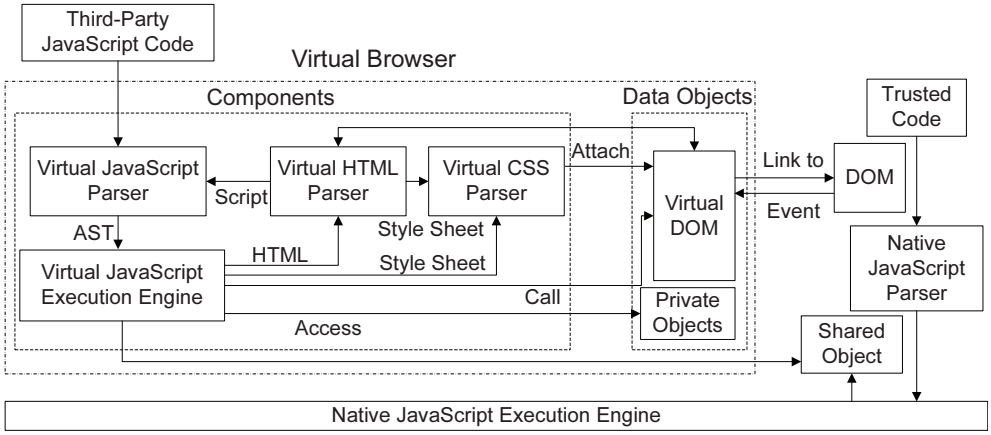

Figure 2.4. System Architecture

virtual JavaScript engine is similar to the one of the native JavaScript engine. Several works[**47, 67**] have details about the native JavaScript engine's interface.

- *Virtual CSS Parser:* It parses CSS codes and attaches the results to Virtual DOM.
- *Virtual HTML Parser:* It parses HTML codes provided by other components and output DOM tree.

Data objects of Virtual Browser include virtual DOM and other private objects.

- *Virtual DOM:* It is linked to the native DOM as an `iframe`. The link is purely for virtual DOM to be shown on the screen. JavaScript access to native DOM from third-party codes is forbidden by our virtualization technique as shown in Section 2.2.3.1. JavaScript access to virtual DOM from trusted codes is also forbidden by `iframe` isolation[3]. Meanwhile, the native DOM also transfers all the events generated automatically by native browsers back to Virtual Browser.
- *Private Objects:* Private data is used to store JavaScript objects that is only accessible to third party JavaScript in Virtual Browser. Section 2.2.3.1 gives the isolation details.

[3]Notice that the isolation provided by `iframe` is purely for preventing access to virtual DOM from trusted code by mistake so that privilege escalation can be minimized (Please refer to Section 2.2.3.1 for details). Virtual Browser is still more robust to unknown native JavaScript engine vulnerabilities than native sandbox approaches, like AdJail [**148**].

Flows. We are introducing possible flows below.

- **Flows inside Virtual Browser.** When a third-party JavaScript code runs into Virtual Browser, the virtual JavaScript parser will first parse it to an AST tree and give the tree to the virtual JavaScript execution engine. The virtual JavaScript execution engine will execute the AST tree similar to a normal JavaScript interpreter does. When HTML content is found, virtual JavaScript execution engine will send it to the virtual HTML parser. Similarly, JavaScript codes and CSS style sheets will be sent to the virtual JavaScript and CSS parsers. Virtual HTML parser will parse HTML and will send scripts/style sheets to the virtual JavaScript/CSS parsers. All of these processes are shown in Figure 2.4. We will give a detailed analysis on these flows in Section 2.2.3.2.
- **Flows between Virtual Browser and Trusted Codes.** Virtual Browser is isolated from trusted codes as analyzed in Section 2.2.3.1. The only flow left is a shared object[4] that connects trusted codes running upon a native browser and third-party codes running on a Virtual Browser.

2.2.2.2. Examples for Several JavaScript Operations. In this section, we illustrate several JavaScript operations in Virtual Browser to show how Virtual Browser works. Some of them are not supported by previous approaches.

with. *with* is a notoriously hard problem in this area. None of the existing works can solve this problem. In our system, *with* becomes quite simple because Virtual Browser interprets JavaScript. For example, as shown in Figure 2.5(a), *with exp* in our system is just a switch of current context.

eval. *eval* is often disallowed by existing approaches totally or partially because it will introduce additional unpredictable JavaScript. As shown in Figure 2.5(b), Virtual Browser just needs to redirect contents inside *eval* back to our virtual JavaScript parser. No matter how many *eval*s

[4]Object here is an abstracted concept, which can also be a single value. According to some recent work done by Barth et al.[**64**], in some cases, values might be less error-prone than objects.

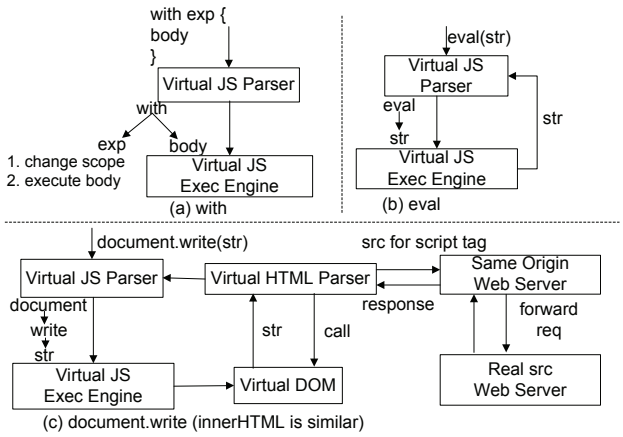

Figure 2.5. Securing Several Dynamic JavaScript Operations

are embedded, such as *eval(eval(...(alert('1'))...)*, JavaScript is still executing inside our virtual JavaScript engine.

document.write/innerHTML. *document.write* and *innerHTML* are related to the HTML parser. As shown in Figure 2.5(c), when virtual JavaScript execution engine encounters functions/variables like these, it will redirect them to the virtual HTML parser by calling methods in virtual DOM. If scripts like $< script\ src = ".." >$ are found in those HTML codes, an XMLHTTPRequest like `http://www.foo.com/get.php?req=...` will be sent to the same origin web server (`www.foo.com`, where all third-party and trusted codes and Virtual Browser are fetched) due to same-origin policy restriction and then redirected to the real web server. JavaScript contents will be received and redirected back to virtual JavaScript parser.

arguments. *arguments* are implemented inside a function. Arguments of the current function are stored in the current running context. When the third party codes use *arguments*, Virtual Browser can fetch them directly.

2.2.3. Security Analysis

In this section, we analyze the security of Virtual Browser by making sure third-party JavaScripts and their data flows only inside Virtual Browser, and create necessary communication channels to the data and resources outside Virtual Browser. We adopt two methods to ensure security : avoidance and redirection. In Section 2.2.3.1, we sandbox all the components inside Virtual Browser by cutting off the inflows and outflows to and from the sandbox. We avoid using some of JavaScript's dangerous functions in the Virtual Browser implementation to achieve isolation. In Section 2.2.3.2, we enable shared objects and communications with security access control. Because we have already built an isolated sandbox, in the second part of the design, we mainly redirect dangerous flows within the third party code back to the sandbox to facilitate communication.

2.2.3.1. Isolation through Avoidance. As we mentioned before, we design our sandbox as another browser built on top of existing browsers, which we call the Virtual Browser, similar to the concept of a virtual machine. The right part of Figure 2.2 shows a Virtual Browser upon a native browser. The Virtual Browser and the trusted JavaScripts from the web site are all running on the native browser. The third-party JavaScript is running on the virtual browser. The method for building the Virtual Browser is similar to building a native browser. We need to build a JavaScript engine, HTML parser, CSS parser and so on. Those components are fully written in JavaScript. We will not focus on how to implement each of the components here, which are not very different from writing a native browser. What we are interested in is how to isolate the Virtual Browser from native browsers. Since we have not introduced communication yet, we only need to enforce isolation between the trusted JavaScripts from the web site and the third-party JavaScripts.

Cutting off Outflows of Virtual Browser. Cutting off outflows of Virtual Browser means that we want to prevent the third-party codes that run on Virtual Browser from running directly on the

native browser. We ensure the third-party codes are trapped inside Virtual Browser. To achieve that, we have the following assumption.

Any JavaScript code has to be parsed in the native JavaScript parsers before it can be executed in native browser.

The Virtual Browser treats a third-party JavaScript as a string, and the string is the input for a virtualized browser instance. Virtual Browser calls the virtual JavaScript parser, which is part of the virtual JavaScript engine, to parse the string, and then executes the code on the virtual JavaScript interpreter. We need to prevent any part of the string from feeding into the native JavaScript, CSS, and HTML parsers. On native browsers, the operations that can cause JavaScript/CSS/HTML parsing are limited. Therefore, we have to avoid using all kinds of operations, such as *eval*, *document.write* and so on when implementing our system, so that the native parsers have no chance to be triggered on the string that contains the third-party JavaScript code.

The follow-up question is how we can find these operations that cause native parsing. We use two approaches: looking up references and looking at the source code of native browsers. Most browsers have their own references and many browsers obey certain standards such as DOM, CSS, and JavaScript. We look at those manuals to figure out which functions trigger native parsing.

However, those manuals may be erroneous and may be lacking details. Call graph analysis on the source code of native browsers is another option. We examine parts of the call graph in which only functions which call the *parse* function will be listed (the functions, which indirectly call the *parse* function through other functions, are also included). We avoid using all of the functions that are direct or indirect callers to native parsing when implementing Virtual Browser.

Cutting off Outflows to the Native JavaScript Parser We need to make sure the third-party codes running in the Virtual Browser cannot leak to the native JavaScript parser. First, we looked up the JavaScript reference[**24**] and verified that only *eval* and function construction from string,

such as *new Function(string)* can result in calling the JavaScript parser. We also did a static call graph analysis of the WebKit[5] JavaScript Core using Doxygen[**13**].

In WebKit, we found that the following functions could call the JavaScript parser directly or indirectly. We trace functions only in the JavaScript Core.

- *opcode op_call_eval.* When *eval* is evaluated, *op_call_eval* is called, and the JavaScript parser is invoked to parse the parameter of *eval*.
- *evaluate.* This function is used by JSC (a test module in the JavaScript Core, not used in real browsers) and the DOM to parse <script> tag, which is outside the JavaScript Core.
- *constructFunction.* It is the function constructor. When feeding a string into the function constructor, JavaScript parsing is triggered. It is used by JSEventlistener which binds a JavaScript function as an event handler for a specific event, and also is used by JSObjectMakeFunction, an open API provided by the JavaScript Core.
- *functions in JavaScript Debugger.* Virtual Browser is not using debugging mode.
- *functions in JavaScript Exception Handling.* A JavaScript string may be reparsed during exception handling in order to get information such as line number, exception id, etc. In the reparsing function, it will reparse exactly the same string as before.
- *numericCompareFunction.* It uses the JavaScript parser to parse a numeric constant string. Numeric constants can be considered as safe, so this is not an issue.

Because the code of Virtual Browser uses only the basic functions in the native JavaScript engine, to avoid using *eval* and function construction through string is enough to prevent third-party JavaScripts in Virtual Browser from leaking out to the native JavaScript parser.

[5]Some undocumented and non-standard functions that can cause parsing may be used in close-source browsers, like IE, and new features may also be introduced in future version of those browsers. However, because we do not even know those new, undocumented, or non-standard functions, they are definitely not used in the source codes of Virtual Browser.

Cutting off Outflows to the Native HTML Parser and the CSS Parser Similar to cutting off flows to the native JavaScript parser discussed above, we need to make sure that third-party HTML or CSS do not leak to the native parsers. We found that the native core JavaScript engine does not call the native HTML or CSS parsers at all, so we can be sure that the third-party HTML or CSS is not leaked to the native parsers from Virtual Browser.[6]

In conclusion, outflows of Virtual Browser are cut off.

Cutting off Inflows of Virtual Browser. Cutting off inflows of Virtual Browser means preventing the trusted JavaScripts of the web site from accessing the objects in Virtual Browser directly. Although the JavaScripts from the web site may not have intentionally malicious behavior, it may influence the virtualized browser by mistake. For virtual DOM, we link data in virtual DOM to an *iframe* tag to prevent trusted codes from the web site to access it by mistake. For other objects, we perform an encapsulation of the *virtualized browser* based on the encapsulation of *Object* in object-oriented languages. We provide only a limited API as the interface and put all other variables and objects as private objects inside Virtual Browser. We also use anonymous objects to prevent inheritance and misuse of Virtual Browser. An example follows.

```
(function(){ this.evaluate= function (){codes} other codes})();
```

From the perspective of the native JavaScripts, they can only see *evaluate* but no other private objects inside the virtual JavaScript engine. The web developer (writing trusted scripts owned by the web site) would be required to avoid overwriting this narrow interface of Virtual Browser.

2.2.3.2. Communication through Redirection. In Section 2.2.3.2, we will discuss data security that makes sure general data is secured, and script security that is more complicated to handle. Then, in Section 2.2.3.2, we give a detailed analysis on each component of Virtual Browser to show how data and scripts are flowing.

[6]Note that functions such as *document.write* belong to DOM and not the JavaScript engine.

Data Security. Data security consists of general data security and security of special data—script.

General Data Security Data in our system is classified into two categories: private data and shared data. Private data refers to virtualized objects and functions in Virtual Browser, which are generated by third-party codes. Shared data refers to some shared objects for communication between the third-party and the trusted codes. Private data in Virtual Browser is safe because of encapsulation of our sandbox in Section 2.2.3.1. We however need to consider the security of shared objects. Data in shared objects can be secured in various ways. We illustrate two methods: mirrored objects and access control.

- *Mirrored Object.* Some objects, such as *String*, *Date*, and *Math* are not necessarily shared. We can create a copy of such an object, which means we turn them into private data. Below is a simplified example copied from Mozilla Narcissus JavaScript engine[**127**].

```
function String(s) {
        s = arguments.length ? "" + s : "";
        if (this instanceof String) {
            this.value = s;
            return this;
        }
        return s;
    }
```

When third-party scripts try to access a property of *String*, third-party scripts are accessing the mirrored *String* defined in Virtual Browser. Similarly, *Object* may be mirrored to prevent using prototype chain to access the original (unique) *Object* object. Notice that not only *Object* is virtualized here but also the whole process of accessing. Because each time third-party codes

call the property *prototype*, they will go through *getValue* interface of Virtual JavaScript engine. Virtual Browser will fetch virtualized contents for it. So the whole accessing process is secured.

- *Access Control.* Virtual Browser uses access control to secure must-share data, which means third-party scripts can access an object only when they have the right to do that. There are many existing works about how to facilitate communications between different entities, such as methods in postMessage channel [**66**], and Object Views [**124**]. They employ different policy engines. We can adopt either of them. Object Views [**124**], which can also be deployed on Google Caja [**94**], is a good choice for us.

Script Security Scripts are also a special kind of data, but they can be executed. Execution of third-party scripts outside of the virtual JavaScript engine may cause the third-party JavaScript to escalate its privileges. Based on the assumption about native execution we made earlier, i.e., scripts have to be parsed before execution, we need to prevent third-party data from flowing into the native parser. Therefore, our task is to track flows that might go out of the virtual JavaScript engine. Scripts in Virtual Browser are classified into two categories: confirmed scripts and potential scripts.

First, some types of data that we know are JavaScript codes, such as the data assigned to *innerHTML*, the parameter of *setTimeout*, etc. For this type of scripts, we use redirection to redirect these kinds of data back into Virtual Browser. There are two categories.

- *Scripts that need immediate execution.* For example, when some JavaScript strings passed to *eval*, they will be added to the JavaScript execution queue immediately. When processing *eval*, we directly put these codes back into our system. For example, *eval(str)* will be interpreted as *evaluate(str)* in which *evaluate* is part of our system.
- *Scripts that need delayed execution.* Some scripts' execution is triggered by certain events or functions. For example, when *setTimeout("alert(1)",100)* is executed, the code inside will be executed 100 milliseconds later. We adopt a pseudo-function pointer here. Virtual Browser

first parses scripts and put it into an anonymous function. In this function, Virtual Browser executes parsed scripts with correct scope and registers this anonymous function with events that will trigger the original function. Therefore, when an event triggers this function or a function calls this function, this function will execute the parsed data inside our system. We still ensure third-party scripts are running in the sandbox. For the aforementioned example, it will look like *setTimeout(function(){execute(parsed_node, exe_context)},100)*. We call this method the *pseudo-function pointer* method.

Second, we do not know if some data is a script or not. This data is therefore a potential script. Potential scripts exist because the trusted JavaScripts running on the native JavaScript engine, plugins, etc. have higher privileges and we do not have control over them. They may get data and execute it as a script, which is the well-known privilege escalation problem. Trusted JavaScripts from the web site will not intentionally behave maliciously, but they may unintentionally grant escalated privileges to the third-party JavaScripts. Next, we will analyze the potential privilege escalation possibilities in our system, which can be classified into three categories.

- *Access Shared Object.* Trusted JavaScript code may access shared objects, which are contaminated by third-party JavaScript code. Therefore, trusted JavaScript code might unintentionally grant escalated privileges to a third-party JavaScript.
- *Access Sandbox.* Trusted JavaScript codes may access the sandbox directly. We have already discussed this issue in Section 2.2.3.1.
- *Access Third-Party JavaScript.* Trusted JavaScript code may want to invoke some third-party JavaScript code. Because of our encapsulation of Virtual Browser, direct access is prohibited. However, if a function call is necessary, Virtual Browser can use pseudo-function pointers discussed earlier to let trusted JavaScript invoke third-party code while ensuring security at the same time.

Because the last two types of privilege escalation are solved by encapsulation, only the first one remains. This type of privilege escalation is not the focus of our paper. We can leverage the solutions from previous works. Finifter et al. [**93**] propose a solution by limiting interface of shared objects and third-party programs in ADSafe. The interface of Virtual Browser is similarly designed to be narrow in order to control the flow.

Security Analysis of Data Flows in Virtual Browser Components. Now, we discuss the flow of data in Virtual Browser. We analyze data flows and script flows among three components, which are the virtual JavaScript engine, the virtual HTML and CSS parsers.

Securing Data Flows of the Virtual JavaScript Engine A narrow interface will help limit data flows and ease data flow examination. As mentioned in Section 2.2.2.1, the interface of the virtual JavaScript Engine has three parts: *putValue*, *getValue* and *function call/return*. Every flow from the virtual JavaScript engine needs to go through this interface.

We show all possible data and script flows of the virtual JavaScript engine in Figure 2.6. Every flow that goes into the native JavaScript engine is conservatively considered malicious because we have no control over the native JavaScript engine. All the flows in Figure 2.6 are presented below.

- *Redirection of Possible Accesses (Flows 1 and 2).* Flow 1 exists because JavaScript can generate JavaScript. For example, *eval* can execute a string as JavaScript. Flow 2 is caused by the fact that some objects, functions and properties may lead to HTML, JavaScript, and CSS parsing. For example, the *document.write* function and *innerHTML* property cause HTML parsing. The *onClick* property causes JavaScript parsing. We use redirection to redirect these functions to corresponding components in our system. For example, a modification of *innerHTML* in virtual DOM node will use the virtual HTML parser in Virtual Browser to parse it.
- *Privilege Escalation (Flows 4, 5, and 6).* Because trusted JavaScript has higher privileges, privilege escalation can happen. It can do anything as we mentioned in Section 2.2.3.2. Flow 4 is

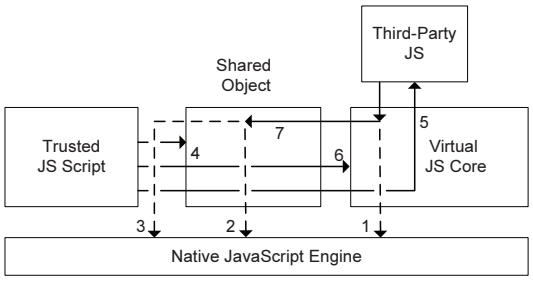

Figure 2.6. Securing Data/Scripts Flows (I)

to access shared objects. Flow 5 is to access third-party JavaScript codes. Flow 6 is to access Virtual Browser itself. Flow 5 is secured by *pseudo-function pointers*. Flow 6 is secured by encapsulation. Flow 4 is hard to secure. We illustrate some methods in Section 2.2.3.2. In this regard, we are on par with state-of-the-art approaches, such as ADSafe [**2**], Web Sandbox [**126**] and Google Caja [**94**].

- *Hidden Access to Native Code (Flow 3).* Flow 3 is a hidden flow. In common cases, JavaScript running on the virtual JavaScript engine cannot access native JavaScripts and through them access the native JavaScript Engine. However, the hidden flow can be triggered by two conditions. First, the third-party JavaScripts running on the virtual engine modify shared objects that belong to flow 7 (part of flow 2). Second, native JavaScripts use a shared object that belongs to flow 4. We can break either of these two conditions to prevent this kind of privilege escalation. We have discussed how to break the second condition in Section 2.2.3.2. We will discuss about cutting the first here. If a mirrored object is used, flow 7 is blocked automatically. If access control mechanism is used, a *write* privilege will have to be carefully given to a third-party program. Usually, a *read* privilege is sufficient.

Securing Data Flows of the Virtual HTML and CSS Parsers The interfaces to the virtual HTML and CSS parsers are narrow. The HTML parser and the CSS parser take a string as input and give

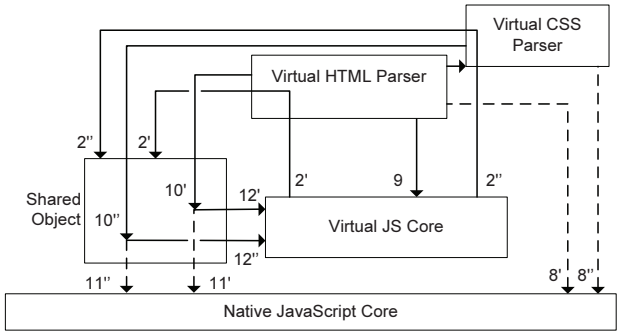

Figure 2.7. Securing Data/Scripts Flows (II)

parsed results as an output. The virtual HTML and CSS parsers can use each other as well as the virtual JavaScript parser. They do not have other interfaces.

In Figure 2.7, we show all the possible flows of the virtual HTML and CSS parsers. We ignore flows that come into the virtual JavaScript engine because they have already been discussed.

Flows $2'$ and $2''$ are redirected from flow 2 in Figure 2.6. Any possible parsing of HTML and CSS is done inside the sandbox. Flow $8'$ and $8''$ are outflows of HTML and CSS parsers, discussed in Section 2.2.3.1. Flow $11'$ and $11''$ are similar to Flow 2 in Figure 2.6. We use *pseudo function pointers* to redirect them to the virtual JavaScript engine (Flows $12'$ and $12''$). Other flows are inside our system and do not cause any security issues. They facilitate communication among components in Virtual Browser.

2.2.4. Implementation

We have implemented a prototype of Virtual Browser. Our virtualized browser contains a virtual JavaScript parser and a virtual JavaScript execution engine, a virtual HTML parser and a virtual CSS parser. We reused and modified the existing JavaScript implementations of the HTML parser[**135**] and the CSS parser[**138**]. For our JavaScript engine, we modified Mozilla Narcissus

metacircular JavaScript Engine[**127**], implemented over JavaScript itself. Moreover, we implemented a simplified version of virtual DOM. Only basic functionality is supported. For example, we support *innerHTML*, *outerHTML*, *innerText*, *document.write*, *document.writeln* and so on in Flow $2'$ (Figure 2.7). As we have already discussed, if some flows are not introduced, it will result only in reduced functionality but no security problems. A production level implementation would also include other components such as an XML parser.

Except the ECMAScript standard, different browsers may implement different non-standard features of the JavaScript language. Our Virtual Browser needs to be compatible with those non-standard features in order to third-party JavaScripts that rely on those features. Two methods are used here. First, we support the standard features and non-standard features which has been implemented by most browsers. For example, we support *try...catch (e) if exp1 ... if exp2 ...* instead of non-standard *try ... catch (e if exp1) ... catch (e if exp2) ....* Second, for important features, we detect browser vendors and versions when necessary. For example, we use *ActiveXObject("Microsoft.XMLHTTP")* in IE but *XMLHttpRequest* in other browsers.

At the same time, sometimes, Virtual Browser needs to provide non-supported features than the underlying native browser. In this case, we need to mimic those non-supported features. For example, IE does not support the keyword *const*; so we use *var* and give it a tag if it is a constant to make it appear like a *const*.

2.2.5. Evaluation

This section is organized as follows. In Section 4.1.5.3, we evaluate the performance of Virtual Browser prototype, memory usage, and parsing latency. In Section 2.2.5.2, 2.2.5.3, and 2.2.5.4, we evaluate Virtual Browser prototype with existing browser quirks and native JavaScript engine bugs, and completeness of our prototype.

2.2.5.1. Performance Evaluation. We measure the execution speed of Virtual Browser with mircrobenchmarks and macrobenchmark, and follow with discussion.

Microbenchmarks We compare the execution speed of Virtual Browser and Microsoft Web Sandbox [**126**], a state-of-the-art runtime approach for sandboxing third-party applications. Microsoft Web Sandbox[**126**] implemented by Microsoft Live Labs as a web-level runtime sandbox written in JavaScript. The idea was derived from BrowserShield[**132**] project, which rewrites dynamic web content and inserts runtime security checks. Web Sandbox has some problems with the virtualization of local variables[7]. Therefore, we only use global variables in our benchmarks for the comparison with Web Sandbox. We however note that since Virtual Browser virtualizes all variables, including both global and local, there are no performance impacts if we use local variables instead of global variables in our approach.

Our experiments are performed on Firefox 3.6. The results of the execution speed of Web Sandbox and Virtual Browser on JavaScript microbenchmarks is shown in Table 2.2. In this experiment, we measure each important atomic operation 10K times and report the cumulative delay. Our system and Web Sandbox achieve nearly the same performance. Microbenchmarks 1-4 are some basic JavaScript operations. Virtual Browser is faster than Web Sandbox for array operations while for arithmetic and functional operations, the two perform nearly the same. Microbenchmarks 5-7 are object operations and have comparable performance on these two. Microbenchmarks 8-9 are string operations. Virtual browser uses mirrored string object which makes it slightly slower.

[7]For example, in the following JavaScript code,
var tempReturn; for (var i = 0;i<10000; i++) tempReturn = fncTest();
Web Sandbox will rewrite the code as follows.
var tempReturn; for(var i = 0; i < 1e4; i++) h(); tempReturn = j.fncTest()
Web Sandbox does not wrap up *i* in the above code. To the best of our knowledge, Web Sandbox tries to avoid wrapping local variables to improve performance. Local variables are supposed to be put in their virtualized environment (Virtualizing local variables should be achieved in virtualized environment but not during virtualized execution). Such relaxations are dangerous because it exposes local variables directly to the native JavaScript engine and because the server and client may have different interpretation of local variables due to browser quirks as we have already mentioned in Section 3.1.1.

Table 2.2. 10K Atomic Operations' Speed Comparison of Web Sandbox and Virtual Browser

Operation	Web Sandbox	Virtual Browser
1. Arithmetic Operation	492ms	480ms
2. Invoke Function	515ms	525ms
3. Populate Array	1019ms	621ms
4. Read Array	1610ms	1217ms
5. Get member on user object	589ms	416ms
6. Set member on user object	566ms	500ms
7. Invoke member on user object	605ms	523ms
8. String Splitting	466ms	532ms
9. String Replacing	475ms	577ms
10. DOM Operation getElementById	707ms	757ms
11. DOM Operation createElement	673ms	822ms
12. With Statement	N/A	333ms
13. Eval Statement with Simple String	N/A	2867ms

Operations	Time
New Game	50ms
Drop a Piece	18ms
Mouse Move	1ms
Game Over	8ms

Figure 2.8. Delays of Operations in Connect Four

Microbenchmarks 10-11 are DOM operations. Because Virtual Browser enforces a check on each element passed to or returned by DOM functions, it is slightly slower. Microbenchmark 12 and 13 evaluates the *with* and *eval* statements which are not supported by Web Sandbox.

Macrobenchmark We measure Connect Four, a JavaScript game from the front page of a popular JavaScript game web site [**23**], which is listed as the first site by Google search when searching for JavaScript games. The task of Connect Four is to connect four pieces of the same color. Delays of operations in Connect Four are shown in Figure 2.8. Virtual Browser does not cause any user visible delay. Due to the fact that *eval* is used, Connect Four cannot be run on Microsoft Web Sandbox without modification.

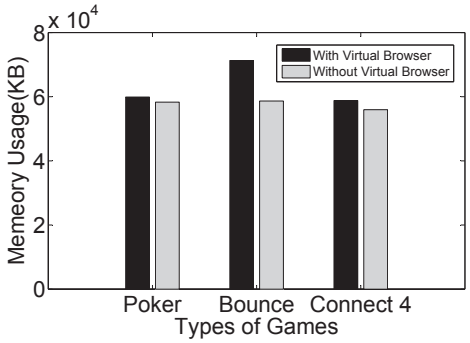

Figure 2.9. Memory Usage

Discussion While Virtual Browser is as slow as Web Sandbox, it is much more secure and complete. Although slowness may limit applicability of this approach, our system remains well suited for securing third party JavaScripts. On average, a single JavaScript operation of a parsed AST tree in our system costs 0.03-0.05ms. For a common JavaScript program without mouse movement events (mouse move events are equivalent to the heavy animation in the later discussion), a user may trigger an event every 500ms, which means one can still write 5000 lines of code (1-2 operations per line). This is long enough to implement a decent program. For a heavy animation script, an image may need to be updated every 50 ms, allowing developers to perform 500 operations to deal with one event, which is long enough for most usages. For example, the *bounce game* in Figure 2.9 is a medium-sized game (800 lines of codes). From the user's perspective, she can feel only a little additional delay. Our system will not be able to support larger animation JavaScripts. However, for common third-party JavaScripts which do not involve very frequent events and are also not very large, the comparative slowness of our system should not pose a problem at all. We believe the situation is similar to Java and Python being preferred for some applications than C and C++ owing to their ease of programming despite their not being efficient enough. Virtual Browser provides an easy way to handle all kinds of third-party scripts, even those using complex and unsafe functions like *eval* and yet without any security glitches.

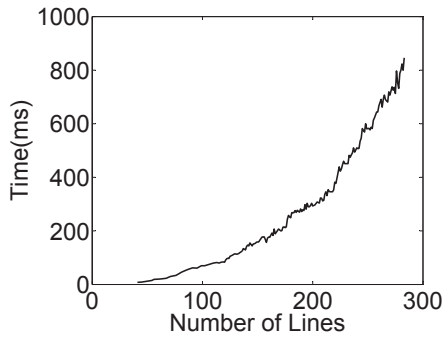

Figure 2.10. Latency of JavaScript Parsing

Memory Usage. Figure 2.9 compares memory usage of third-party JavaScript applications running in Virtual Browser to that when they are running in a native browser. We choose third-party JavaScript games from a popular web site[**23**]. Connect 4 and poker are two games with user interaction. Users need to move and click the mouse to play these games. These two games' memory usage with Virtual Browser is about 2% higher than that without Virtual Browser. Bounce is a game with substantial animation and user interaction in which the user can see a ball bouncing in the screen. Bounce takes about 20% more memory when running in Virtual Browser. This is because high animation programs require more resource from Virtual Browser.

Parsing Latency. We measure parsing latency of the virtual HTML and JavaScript parsers in Figure 2.10 and Figure 2.11 by parsing a game web site [**23**] and a JavaScript game, *Connect 4* from it. Both of these parsers are written in JavaScript. In Figure 2.10, JavaScript parsing rates decrease slightly when the number of lines increases (as shown in [**33**], top-down parsing is a polynomial time process). As seen in Figure 2.11, HTML parsing is faster than JavaScript parsing because JavaScript language with more types of AST nodes is more complex than HTML.

The JavaScript parser written in JavaScript is not very fast. An alternative of this virtual JavaScript parser would be to pre-parse first-order JavaScript and HTML code at server side, generate a JavaScript Object (parsing results) that our execution engine requires, and transmit JSON format

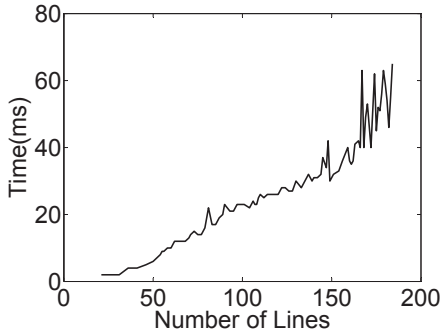

Figure 2.11. Latency of HTML Parsing

to the client. JSON parsing speed is fast enough at about 600K/s. However, this may be vulnerable because the JSON generator at the server side and the JSON parser at the client side may interpret JSON differently. Due to the simplicity and well-formatted-ness of the JSON protocol, the chance of different interpretations is low; so we can still consider this approach as an alternative. We still execute third-party JavaScript on the virtual JavaScript engine (and not on the native JavaScript engine) and even if we adopt this alternative, our approach will still be more secure than other runtime approaches.

Unlike first-order JavaScript, high-order JavaScript and HTML (generated by scripts dynamically), like the scripts introduced by *innerHTML* and *document.write*, have to be parsed in the parser written in JavaScript. Compared to first-order JavaScript, the amount of these kinds of codes is relatively small. Thus, using our parser will not introduce too much delay and the JSON approach remains viable.

2.2.5.2. Browser Quirks Compatibility. Since we parse scripts only once at the virtual JavaScript engine, and do not rely on the native parsers, we are not vulnerable to browser quirks. Additionally, scripts cannot leak from our system to the native JavaScript engine. We evaluate our system with 113 browser quirks' examples listed in the XSS Cheat Sheet [**51**]. The XSS Cheat

Sheet contains mostly examples of XSS attacks that are caused by browser quirks. The results show that none of the 113 browser quirks lets third-party JavaScript codes bypass the virtual JavaScript engine, regardless whether the language features are supported by Virtual Browser.

2.2.5.3. Robustness to Unknown Native JavaScript Engine Bugs. We evaluate the robustness of Virtual Browser to unknown native JavaScript engine bugs. In this experiment, we use an old version (before 2009) of Firefox. All the bugs (14 in totals) recorded in CVE [**7**] related to SpiderMonkey JavaScript engine in the year of 2010 and 2009 are evaluated. The results are the same as we discussed in Section 2.2.1. Running of example exploits in the database does not trigger those vulnerabilities in Virtual Browser.

Furthermore, our implementation of Virtual Browser does not satisfy the preconditions for triggering any of those vulnerabilities. We illustrate three as examples. Others are similar.

- **CVE-2010-0165:** Native *eval* is required to trigger this vulnerability. Virtual browser source code does not use native *eval* statement.
- **CVE-2009-2466:** The vulnerability is triggered by the statement $with(document.all)\{\}$. Virtual browser source code does not use *with* statement.
- **CVE-2009-1833:** The prototype of an object is set to be null inside the prototype function. Virtual browser source code does not use prototype in this way.

Discussion. **How does Virtual Browser deal with bugs in Virtual JavaScript engine?** The experiment of this section is performed to prove the hardness of circumventing virtual JavaScript engine to exploit native JavaScript engine. The security of virtual engine is fully analyzed in Section 3.1.4. Moreover, virtual JavaScript engine of Virtual browser is written in JavaScript, a type safe language that does not have vulnerabilities like buffer overflow in native JavaScript engine.

Where does the robustness come from? Virtual Browser is implemented by a type safe language, which gradually reduces the number of possible vulnerabilities. However, the enhanced security does not only come from type safe language but also the virtualization technique.

Browser virtualization adds another layer that increases security assurance. In particular, Virtual Browser utilizes virtualization to isolate JavaScript codes. Attackers need to break Virtual Browser first and then native browser to steal information. Similar to a virtual machine that has higher security assurance than other native sandboxes, Virtual Browser with another virtualization layer has its advantages.

2.2.5.4. Completeness of Virtual Browser Implementation. We evaluate the completeness of our prototype of Virtual Browser in order to show that our other experimental results are convincing. We use test cases of ECMA-262 Edition 1 from Mozilla [**25**]. The results show that we can pass 96% of the test cases. For some categories, such as *JavaScript Statement*, *String*, *Expressions*, *Types*, etc., we can pass 100% of the test cases. The worst of all is Object Objects, for which we can only pass 72% test cases. The incomplete implementation will only affect the functionality of Virtual Browser but not the security of Virtual Browser because we introduce the data flows after isolation. Security is always the most important concern in Virtual Browser.

When we are trying to run the same test in Web Sandbox [**126**], Web Sandbox cannot even run the test itself because Web Sandbox does not have full support of *eval*. However, *eval* is required in the driver of these test cases. This also proves the importance of *eval* without which we cannot perform the tests. Therefore, we have to give up running the test for Web Sandbox. Our manual check shows that they do not have complete implementation either.

2.3. Detection of Contents Leaked from Principals (JShield)

In the past few years, drive-by download attacks exploiting browser vulnerabilities have become a major venue for attackers to control benign computers including those of reputable companies. For example, in February 2013, both Facebook and Apple confirm being hit in "watering hole attack" [**4**], a variance of drive-by download attack. In such an attack, the attacker compromises a web site commonly visited by victims, injects drive-by download attacks, and then waits for victims to come, just as a predator sitting at a water hole in a desert for prey.

To defeat these severe attacks, many anomaly-based approaches [**74, 87, 136, 84**], which tune the detection engine based on attacker-generated exploits and benign web sites, have been proposed and significantly improved state of the art. However, anomaly-based approaches could succumb in adversarial environments. Inspired by the data pollution on polymorphic worm detection [**130**], we designed and implemented polluted drive-by download attacks, which could significantly reduce the detection rates of those anomaly-based detectors.

To countermeasure such data pollution, we turn to vulnerability-based approaches. In this emerging direction, although the precise description of a network protocol layer vulnerability signature has already been used in the network intrusion detection systems (NIDS) [**154, 115**], its application on Web drive-by download attack detection is very limited. In BrowserShield [**132**], Reis et al. rewrite JavaScript codes and enforce stateless security policies (signatures), such as checking the parameter length of a function call. The other work by Song et al. [**141**] matches the inter-module communication with vulnerability signatures. However, neither of these techniques can precisely represent the exact vulnerability conditions of drive-by download attacks. More specifically, the rules in BrowserShield do not have stateful data structures to record control or

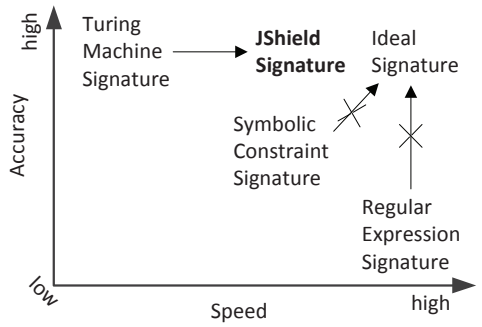

Figure 2.12. Signature Comparison.

data flow, and the signatures in the work by Song et al. lack sufficient information for control flow at the inter-module level.

In summary, we believe that an approach detecting drive-by download attacks should address the following **challenges:**

- **Signatures for Stateful Intra-module Vulnerabilities.** The signature for drive-by download attacks targeting stateful intra-module vulnerabilities should contain both control and data flows during matching.
- **Resilience to Polluted Data.** The system should be resilient to polluted training and testing samples provided by the attackers.
- **High Performance.** The runtime detection system should have an acceptable overhead to the pure execution of web pages.

In this paper, we propose JShield, a vulnerability-based opcode level detection of drive-by download attacks. JShield has been adopted by Huawei, the world's largest telecommunication company. We position JShield opcode signatures in Figure 2.12 with other known vulnerability signatures. Neither symbolic constraint signature nor regular expression signature can represent a drive-by download vulnerability that takes Javascript, a Turing complete language, as input, since they do not have loops. Meanwhile, a traditional Turing machine signature is so complex that it requires a signature as large as a whole browser. Thus, we abstract lower level Turing machine to a higher level opcode Turing machine and design detection format of our opcode signature. To

further make it scalable, we also use regular expression (filter format of opcode signature) to filter a large number of benign traffic.

The current signature generation process involves some manual effort. However, we believe that the amount of manual work is small due to the small number (approximately 10) of vulnerabilities each year. Actually, even for a large amount of signatures, most network intrusion detection/prevention system (IDS/IPS) vendors, such as Snort, Cisco and Juniper, all generate these signatures manually [**41, 6**].

Besides being the $first$ to design polluted drive-by download attacks and evaluate their effectiveness on the state-of-the-art anomaly-based approaches, we make the following contributions in this paper:

- **Stateful Drive-by Download Vulnerability Signatures.** Our vulnerability signature is a deterministic finite automaton (DFA) with a variable pool recording both control and data flows to detect stateful intra-module vulnerabilities of drive-by download attacks.
- **Vulnerability Signature at the Opcode Level.** A vulnerability signature at the opcode level can precisely describe a given vulnerability.
- **Fast Two-stage Matching.** We design a two-stage matching process to speed up the detection. The filtering stage adopts fast regular expression matching for a given test sample, and then if the sample is captured at the filtering stage, it is subject to a further matching with opcode signature.

To evaluate JShield, we investigate the vulnerability coverage of JShield and find that JShield is able to handle all the recent JavaScript engine vulnerabilities of web browser and portable document files (PDF) reader. The overall evaluation shows that JShield has introduced little performance overhead to pure execution. The average overhead of top 500 Alexa web sites is 9.42% and the median is 2.39%.

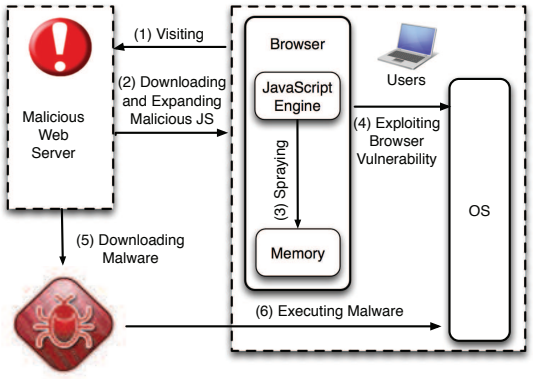

Figure 2.13. Six Steps of a Drive-by Download Attack.

2.3.1. Threat Model

The paper focuses on drive-by download attacks consisting of two major stages: pre-exploit stage and post-exploit stage, which can further divide into six steps as shown in Figure 2.13.

At the pre-exploit stage, a benign user is lured to visit a malicious web site (step one). Then, malicious contents are downloaded, and malicious JavaScript codes, possibly obfuscated by eval, setTimeout, and DOM events, get expanded by execution (step two). During execution, some malicious JavaScripts also fill the heap with shellcodes and nop sleds to overcome address space layout randomization and facilitate attacks (step three).

After all the preparation, the malicious JavaScript exploits a certain vulnerability (step four), and thus the injected shellcode takes control of the browser to download malware (step five), which is subsequently executed on the victim machine (step six).

To distinguish our threat model from others, we also mention other attacks below - these however are *out of scope* of the paper.

Attacks without any JavaScript Interaction. Similar to existing works like Zozzle [**87**], we only focus on the JavaScript part of a drive-by download attack. If an attack is purely related to HTML parser, CSS parser or font processing, neither Zozzle with abstract syntax tree (AST) features nor JShield with opcode vulnerability signatures can detect the attack. Instead, due to lack of full featured obfuscation techniques, this type of attacks should be prevented by traditional NIDS.

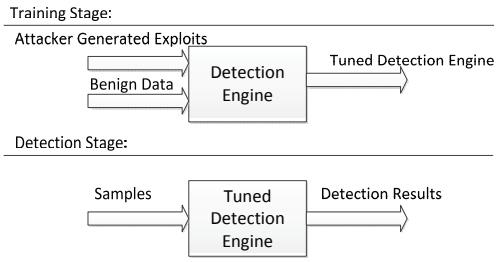

Figure 2.14. Anomaly-based Approaches to Detect Drive-by Download Attacks.

Other Web-based Attacks. Other web-based attacks, such as cross-site scripting (XSS) attacks, cross-site request forgery (CSRF), and so on are out of scope of the paper. People should rely on existing defense mechanisms [**149, 65**] for those attacks.

2.3.2. Polluted Drive-by Download Attacks

Anomaly-based approaches [**87, 84, 136, 102**] first train a detection engine based on exploits generated by attackers as well as benign samples collected from the Internet, and then perform detection by the tuned engine. An overview architecture is shown in Figure 2.14. Normally, a machine learning engine is deployed in the training stage, which extracts malicious and benign features from training data and then trains the detection engine.

In this section, we discuss the efficacy of anomaly-based detection in adversarial environment. Generally, an anomaly-based detection could be evaded in two ways, namely, polluting attacker generated exploits in training stage and altering malicious samples in detection stage. We introduce data pollution in detection stage first because of its effectiveness. Then, the training stage attack is more complicated and will be explained in our Technical Report [**78**]. For both types of data pollution, we use naive Bayes engine adopted by Zozzle [**87**] as an example to show how to evade anomaly-based approaches.

2.3.2.1. Polluting Samples at Detection Stage. An attacker can alter malicious samples by injecting benign features (BF), thus increasing the probability of classifying those samples as benign and evading naive Bayes engine. Intuitively, the benign features decrease the anomaly by reducing

the significance of the malicious features statistically. Here is the detailed reason. Assume there is a sample with malicious feature (MF). According to the definition, $P(M|MF)$, the probability of malice given the existence of one malicious feature, is larger than 0.5. Now let us assume that one benign feature (BF) is introduced to that file. Given that MF and BF are independently distributed in naive Bayes, we have Equation 2.1.

$$\begin{aligned} P(M|MF,BF) &= \frac{P(MF,BF|M)*P(M)}{P(MF,BF)} \\ &= \frac{P(MF|M)*P(BF|M)*P(M)}{p(MF)*P(BF)} \\ &= \frac{P(BF|M)}{P(BF)}*P(M|MF) \end{aligned} \tag{2.1}$$

Since fewer BFs exist in malicious files, $\frac{P(BF|M)}{P(BF)} < 1$. Thus, Equation 2.1 shows that the existence of one benign feature reduces the probability of the sample's malice. If enough benign features are introduced, the probability of the sample's malice $P(M|MF,BF,BF1,BF2...)$ will be eventually less than 0.5, resulting in a mislead of the Bayes classifier.

2.3.2.2. Real-world Experiments. We use Zozzle [**87**], the most recent and successful machine learning detection, as a case study to show how to evade an anomaly-based approach, however, our discussion is not restricted to Zozzle. Since Zozzle is not open source, we strictly followed what has been described in the paper [**87**], implemented our version of Zozzle, and reproduced comparable detection rate for unpolluted samples as the one reported by Zozzle.

The testing data set is from Huawei, which contains unclassified malicious JavaScript codes collected from their customer reporting, other anti-virus software reporting, etc. After filtering all the landing pages, we collect 880 malicious samples (Zozzle adopts 919 JavaScript malware samples [**87**]). Meanwhile, Top 4000 Alexa web sites [**3**] are used as benign samples during

Table 2.3. Zozzle's Detection Rate.

	Original Rate	Rate after Pollution at Detection Stage
True Positive	93.1%	36.7%
False Positive	0.5%	0.5%

training. After manual and automatic selection documented in Zozzle, we collect 300 benign and malicious features.

Since an attacker cannot acquire our benign features used in the system, we collect all the common features among Top 5,000 Alexa web sites and add them to malicious samples. The size of each malicious sample increases 45% on average, *i.e.*, 396KB. As shown in Table 2.3, pollution at the detection stage decreases the overall accuracy to 36.7%.

2.3.3. Overview

As discussed in previous section, attacker generated samples adopted by anomaly-based detections could be polluted, and thus we resort to vulnerability-based detections. In this section, we first present two types of deployment for JShield. Then, we model drive-by download vulnerabilities based on their control and data flows.

2.3.3.1. Deployment. JShield is a dynamic vulnerability signature based approach to de-obfuscate and detect drive-by download attacks. There are two major types of deployment for JShield: 1) at the Web Application Firewalls (WAF) or Web IDS/IPS and 2) at Web malware scanning services. For the former, JShield is deployed as a component of anti-virus software, or as a detection engine at Internet Service Providers (ISP) gateways. For example, Huawei deploys our system in their intelligent cloud, inspecting potential malicious traffic from their switches and routers. On the other hand, JShield can also be deployed on the sever side as a Web malware scanning service, by search engines such as Google and Bing, or by a security company for online web malware scanning.

Compared to an anomaly based approach like Zozzle [**87**] which needs to retrain the detection engine to accommodate new drive-by download exploits, JShield only needs to update its vulnerability signature database for new drive-by download vulnerabilities. Due to that fact that the number of vulnerabilities is always much less than the amount of exploits, the update overhead of JShield will be small.

2.3.3.2. Vulnerability Modeling. Traditionally, there are three types of signatures in literature [**73**]: Turing machine signature, symbolic constraint signature, and regular expression signature. We look into those signatures in the context of a drive-by download attack where the vulnerable program is a browser and the input exploit is a JavaScript program. The Turing machine signature generation process from Brumley et al. [**73**] would output a signature as large as a browser, hence making it unusable. On the other hand, neither symbolic constraint signature nor regular expression can represent a complex language like JavaScript which have loops. Thus, for preciseness, we need to have a new signature between a Turing machine signature generated by Brumley et al. and a symbolic constraint signature. For matching speed, we first use regular expression signature to filter a majority of benign JavaScript and then detect malicious JavaScript using the precise detection format of our opcode signature. Since regular expression signature is well known, we focus on the detection format of our opcode signature.

To model a vulnerability, we define a vulnerability condition as a function that takes a certain path of a program's control and data flow graph, and output whether the path is exploitable or not. Formally, given $c \in C$, where C is all possible paths of the program's control flow graph, and $d \in D$, where D is all possible paths of the program's data flow graph, a vulnerability condition k is a function,

$k : C \times D \rightarrow \{Safe, Exploit\}$

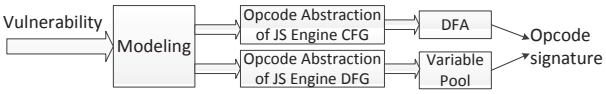

Figure 2.15. Vulnerability Modeling.

In order to match a certain vulnerability, its corresponding vulnerability signature need to match both the path in the control flow graph and the one in the data flow graph. In the context of a drive-by download attack where the vulnerability exist in a JavaScript engine or a plugin, our observation is that control and data flow graphs of the opcode input is an abstraction of the control and data flow graphs of the native JavaScipt engine. Thus, as shown in Figure 2.15, we can model the underline vulnerability in the JavaScript engine or plugin by abstracting the control and data flow to the opcode level.

To study the *control flow*, we investigate the source code of several JavaScript engines [**47, 127, 43**], and find that the main body of a JavaScript interpreter switches to different branches of codes based on the input opcode through a code pattern similar to *select input case opcode*. Then, different opcodes and parameters determine subsequent API calls to external objects, plugin or DOM. Therefore, we form the control flow graph (CFG) of JavaScript engine into an opcode driven three-layer structure as shown in Figure 2.16. The CFG will shift based on the input opcode sequence. In other words, opcode CFG is built upon the original application CFG.

Next, we categorize vulnerabilities into two types: JavaScript engine vulnerability (including a web browser JavaScript engine and/or a plugin JavaScript engine such as Adobe Reader JavaScript engine) and external JavaScript engine vulnerability. We will explain them respectively.

Let us assume that the control flow graph of a JavaScript engine vulnerability condition is triggered by travelling through S1_1 and S1_2 of Figure 2.16. Since there is only one path to travel to S1_1 and/or S1_2, which is to offer the corresponding opcode, the opcode level signature represents this vulnerability.

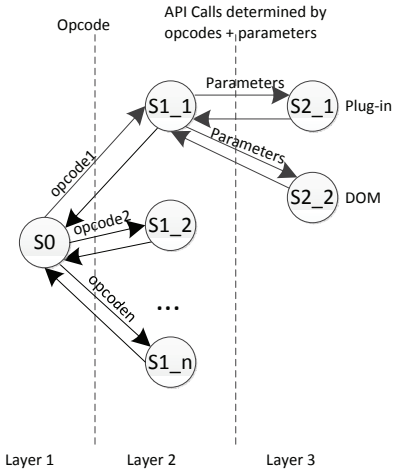

Figure 2.16. Simplified Opcode Level Control Flow Graph of JavaScript Interpreter.

For an external JavaScript engine vulnerability, the API calls to those components such as plugin and DOM are determined by opcode sequences and parameters. Song et al. [**141**] show that the inter-module communication can represent a vulnerability and thus our opcode signature can achieve the same functionality.

Therefore, we propose an opcode level deterministic finite automaton (further explained in Section 2.3.4) to match the opcode control flow, an abstraction of the JavaScript engine control flow, in a vulnerability condition.

To study the *data flow*, JShield needs to record additional states related to the vulnerability. Therefore, we propose a variable pool (further explained in Section 2.3.4) to match the opcode data flow, an abstraction of JavaScript engine data flow, in a vulnerability condition.

Now, we illustrate the point by a concrete running example from CVE database.

Example I. In Figure 2.17, we show how to trigger CVE-2009-1833 in the pseudo code of JavaScript engine. The vulnerability is triggered by two conditions: (i) looking up through prototype chain to get a getter function, and (ii) setting the prototype itself to be null inside the getter function.

When we abstract the JavaScript engine vulnerability to the opcode level and take a look at a concrete exploit example triggering vulnerability in Figure 2.18 and Figure 2.19, we find that

```
1 switch (opcode) {
2   case get_by_id:
     // (1) look up the prototype chain
     // (2) invoke getter method
     // (3) move results to register r0
3         break;
4   case put_by_id:
     // (4) move null to the prototype
5    break;
6 }
```

Figure 2.17. Pseudo JavaScript Engine Codes for CVE-2009-1833.

```
1  var obj = new Object();
2  obj.__proto__.__defineGetter__("a", function () {
3              this.__proto__ = null;
4              gc();
5              return 0;
6              });
7  obj.a;
```

Figure 2.18. Example I: CVE-2009-1833: Malicious JavaScript Exploit that can Trigger a Mozilla Firefox JavaScript Engine Vulnerability.

the opcode level CFG is an abstraction of the underline level JavaScript engine CFG. S1_1 in Figure 2.16 is visited by *get_by_id* and S1_2 is visited by *put_by_id*. For the data flow, to match CVE-2009-1833, JShield needs to remember the memory address of the prototype.

In sum, the JShield signature needs to match both the control flow graph and the data flow graph of opcode sequence of a JavaScript code, an abstraction for the control and data flow graph of the underline JavaScript engine, for a drive-by download vulnerability condition as shown in Figure 2.15.

2.3.4. Opcode Vulnerability Signature

As discussed in Section 2.3.3.2, a successful opcode signature needs to match both the control flow and the data flow of a vulnerability condition. In this section, we introduce the detailed design of opcode signature matching drive-by download vulnerabilities. To speed up the matching process,

```
[ 199] get_by_id    r0, r1, a(@id1)
[   0] enter
[   1] convert_this r-7
[   3] mov       r0, r-7
[   6] put_by_id  r0, __proto__(@id0), Null(@k0)
[  15] ret Int32: 0(@k1)
```

Figure 2.19. Generated Opcode Snippet when Executing JavaScript Codes in Figure 2.18.

two types of opcode signature, the detection format and the filter format, are described here by their definition, structure and matching process. In the end, we present the robustness of opcode vulnerability signature to polymorphic attacks.

In the current version of JShield, all the opcode vulnerability signatures are generated manually for each vulnerability. However, we believe that the amount of involved manual work is small due to the small number ($<$100) of vulnerabilities each year. Actually, even for a large amount of signatures in an intrusion detection system like Snort, those signatures are all generated manually [**41**]. Further, we also expect future improvement can automate vulnerability signature generation. As an analogy, Shield [**154**] proposes protocol level vulnerability signatures, and then Brumley et al. [**73**] propose their automatic generation.

2.3.4.1. Definition. Opcode signature is a signature to match an opcode sequence, an instruction set generated by a JavaScript interpreter for efficient execution. For example, opcodes in Figure 2.19 are the results of transmitting JavaScript code in Figure 2.18. Op code signature has two formats: a detection format used for matching and a filter format used for fast filtering.

We first formalize detection format of opcode signature as a deterministic finite automaton (DFA) plus a variable pool in Definition 1.

Definition 1. *We define detection format of opcode signature as a 10-tuple (Q,* Σ*, P, V, g, G, f,* q_0*,* p_0*, F), where*

- Σ *is finite set of input symbols,* P *is finite set of variables,* V *are the value set of* P*, and* Q *is finite set of states.*
- g*, is a function* $P \rightarrow V$.
- G *is the set of all possible* g.
- *f is a transition function,* $Q \times \Sigma \times G \rightarrow Q \times G$.
- $q_0 \in Q$ *is a start state,* $p_0 \subseteq G$ *is an initial variable pool, and* $F \subseteq Q$.

For input $a_i \in \Sigma$ *and a variable pool* $p \subseteq G$*, the next state of the automaton obeys the following conditions:*

1. $r_0 = q_0, p = p_0$.
2. $< r_{i+1}, p > = f(r_i, a_i, p)$*, for i = 0,..., n-1.*
3. $r_n \in F$.

In Definition 2, we formalize the filter format[8] of the opcode vulnerability signature as a regular expression.

Definition 2. *We define the filter format of opcode signature as a 5-tuple (Q,* Σ*, f,* q_0*, F), where*

- Σ *is finite set of input symbols, and* Q *is finite set of states.*
- *f is a transition function,* $Q \times \Sigma \rightarrow Q$.
- $q_0 \in Q$ *is a start state, and* $F \subseteq Q$.

For input $a_i \in \Sigma$*, the next state of the automaton obeys the following conditions:*

1. $r_0 = q_0$.
2. $r_{i+1} = f(r_i, a_i)$*, for i = 0,..., n-1.*

[8]Unless specified, opcode signature refers to the detection format of opcode signature. The filter format refers to the filter format of opcode signature.

Detection Format:

#	Method	Opcode	Condition	Action	Next
(1)	match	get_by_id	isFromProtoChain()	x=proto	(2)
	default			quit	N/A
(2)	match	enter	true	i=0	(3)
	default			quit	N/A
(3)	match	enter	true	i=i+1	(3)
	match	ret	i==0	quit	N/A
	match	ret	i>0	i=i-1	(3)
	match	put_by_id	x==dst & src==null	report	N/A
	default			jmp	(3)

Filter Format:

```
get_by_id enter .* put_by_id
```

Figure 2.20. Opcode Signature for CVE-2009-1833.

3. $r_n \in F$.

2.3.4.2. Structure. We introduce the structures of the detection and filter format of opcode signatures in this section.

Detection Format. The detection format of an opcode signature, as shown in Figure 2.20, can be formalized into the following three concepts: clause, sentence, and signature.

A *clause* in an opcode signature consists of five fields: "method", "opcode", "condition", "action", and "next". The "method" field specifies what to be taken in this clause, where two methods are currently defined: "match" and "default". "Match" means to match the opcode, and "default" means that the default actions should be taken if no matches are found in other clauses. Then if both "opcode" field matches the input opcode and the expression in "condition" field is true, the action in the "action" field will be taken and current state will be transferred to the number in "next" field, which represents the sentence number that will be explained right after this paragraph.

Multiple clauses plus an index number together build *a sentence*, a state in automaton. The number is used to differentiate one sentence from the others. The clauses in one sentence are in sequence, which means if JShield finds the first match, the remaining ones will be skipped. If no matches are found, the action corresponded with the "default" clause will be taken.

A *signature* consists of multiple sentences. During matching, the automaton will transfer from one sentence to anther based on the matching results.

Filter Format. The filter format of an opcode signature, as shown in Figure 2.20, can be formalized as a regular expression, which takes a series of opcodes as input. For detection and filter format of opcode signature, the following statement holds: "Each detection format of an opcode signature has a corresponding filter format of that opcode signature".

Here is the reason. Given a detection format of an opcode signature, for each sentence, we extract all *Opcode* fields and align them into a unit by bracket symbol of regular expression. Then, by following the jmp operations in *Action* field, we align the units into a regular expression. If a self loop is recognized, a symbol $*$ is introduced. The end of the regular expression is one or multiple opcodes in bracket that leads to the vulnerability.

Data Structure of Both Formats. The filter format is simply stored as a regular expression. To speed up matching process, we construct a reverse index for the detection format of opcode signatures by the opcode field. Suppose we have two signatures: Sig1 and Sig2. Each signature has two sentences. Each sentence has two clauses. Under each opcode, both two signatures exist. Under each signature, both two sentences exist. Under each sentence, only the clause with that opcode is present. Other clauses of that sentence are under other opcodes.

2.3.4.3. Generating Opcode Vulnerability Signature. We generate the opcode vulnerability signatures semi-automatically with the following three steps.

1. Based on the semantics of the vulnerability (*e.g.*, from the CVE description or vulnerability patches), we locate the opcodes that are involved in the vulnerability. We create a DFA with each involved opcode being a node (state).

2. From the data flow part, we extract the critical data structure involved in the vulnerability related to each opcode operation and define a variable in the variable pool of "Action" field in opcode signature.

3. We combine the DFA and the variable pool together by introducing each variable to the "Condition" field of opcode signature based on the data flow connection between opcodes.

Again, we use CVE-2009-1833 in Figure 2.18 as an example. We first automatically generate control and data flow [**108**]. Then, manual work is involved to find out the semantics of the vulnerability, *e.g.*, for CVE-2009-1833, line 3 and line 7 together cause the vulnerability. From the control flow graph part, the sequence of three opcodes (get_by_id, enter, and put_by_id) will lead to the vulnerability condition. On contrary, another sequence of three opcodes (get_by_id, enter, and ret) will lead to a safe state. Therefore, we create a detection format of our opcode vulnerability signature with three states in DFA as shown in Figure 2.20, and meanwhile, we use a counter i to record the number of opcode enter and ret. Next, from data flow part, we find that line 3 and line 7 are connected by the memory address of the prototype, and therefore, we use x in variable pool to record that data. In the end, we combine the DFA and the variable pool to the detection format of our opcode vulnerability signature, and follow steps discussed in Section 2.3.4.2 to generate the filter format of opcode vulnerability signature.

2.3.4.4. Matching Opcode Vulnerability Signature. The matching process of opcode signatures can be divided into two parts: pre-matching by the filter format of opcode and matching by the detection format of opcode signature.

At the filtering stage, we match the opcode sequence outputted from de-obfuscation engine with the filter format of opcode signature. If a sequence of opcodes does not match with the filter format of opcode signature, we drop it off because it will not match with the detection format of

Algorithm 1 Matching Detection Format

```
1: State ← Starting_State
2: for Input_Opcode in All_Opcodes do
3:     for Signature in Pool[Input_Opcode] do
4:         Sentence ⇐ Signature[State]
5:         Clause ⇐ Sentence.Clause
6:         if equal(Method, Match) then
7:             if (IsAllTrue(Clause.Conditions) then
8:                 Take Actions
9:                 Signature.State ⇐ New_State
10:                Break
11:            end if
12:        else[equal(Method, Default)]
13:            Default Actions
14:            Signature.State ⇐ Default_State
15:            Break
16:        end if
17:    end for
18: end for
```

Sample One:

```
this.__defineGetter__("x", function (x){
            'foo'.replace(/0/g,[1].push)
          });
for (let y in [,,,])
   for (let y in [,,,])
       x = x;
```

Sample Two:

```
while (true) {
  Array.prototype.push(0);
}
```

Figure 2.21. Example II: Different Samples that Trigger CVE-2009-0353 (Extracted and Simplified from CVE Library).

that opcode signature either. By filtering a large amount of unmatched samples using fast regular expression operation, we can accelerate the total matching process.

After the filtering stage, we match the opcode sequence with detection format. The pseudo-code of the matching algorithm is shown in Algorithm 1. Given one opcode as an input, the matching algorithm goes over every signature with that opcode. For each signature, JShield directly fetches the corresponding clause that belongs to the sentence of the current state because JShield has already indexed all the signatures by opcodes. Then, JShield checks whether the conditions are satisfied, and accordingly takes actions. The complexity for this process is O(Maximum number of Signatures per Opcode $\times$ Number of Opcodes).

2.3.4.5. Robustness to Polymorphic Attacks. A CVE-2009-0353 example in Figure 2.21, which is triggered when repeating push operation of an array exceeding the memory limit, shows that JShield reduces polymorphic attacks. Instead of reporting an out-of-memory error, illegal memory address will be overwritten.

In Figure 2.21, we show two different snippets of JavaScript triggering CVE-2009-0353, which fire push operation in two different ways at the JavaScript level. However, at the opcode level, both need to call opcode *get_by_id* first to get *push* method, and then repeat using opcode *call* (only one *call* is shown in the figure). After generating the call graph for two JavaScript engines [**43, 43**] by doxygen [**13**] and thus examining functions that calls *push* method, we find that the only way of calling *push* method is through the opcode *call*. In other words, the opcode signature maps to the vulnerability.

For polymorphic attacks, we show that the number of polymorphism reduces at the opcode level because (i) one opcode signature maps to multiple source code representations of that vulnerability; and (ii) one source code representation of a vulnerability maps to one specific opcode signature given an opcode instruction set.

The reasons are as follows. Assume a vulnerability is represented by an opcode signature. We follow the state transition and get opcode sequence as follows: op_code1, op_code2, op_code3 and so on. Each opcode can be included in multiple JavaScript statement. For example, *op_call* can be triggered by direct function call and *getter* property of an object. Similarly, *op_jmp* can be triggered by *while* loop, *for* loop, and so on. We will choose different JavaScript statement and write corresponding JavaScript source code. To the opposite, if we have a source code representation of a vulnerability and the opcode instruction set is fixed, we can feed the source code into the interpreter with the opcode instruction set. One unique opcode sequence is outputted from the interpreter.

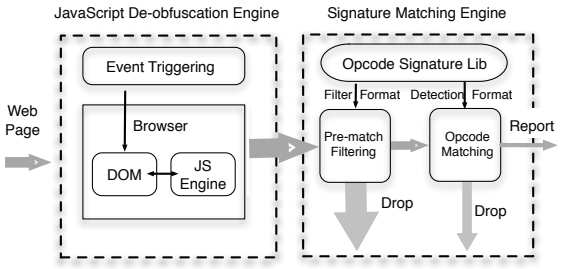

Figure 2.22. System Architecture.

2.3.5. System Architecture

Figure 2.22 shows the overall architecture of JShield, which consists of two main engines, a JavaScript de-obfuscation engine and a signature matching engine. The former takes a web page as input, de-obfuscates JavaScripts, and then outputs the corresponding opcode sequence; the latter takes an opcode sequence as input, matches the sequence with opcode signature, and finally gives a report about whether the incoming web page is malicious .

The detailed process is as follows. When a web page is fed to a JavaScript de-obfuscation engine of JShield, it is executed on a real browser with event triggering module, which mimics user's behaviors to trigger all the DOM events. If the web page contains PDF, JShield adopts MP-Scan [**119**] to hook an Adobe Reader and output all the opcode sequences. After de-obfuscation, the signature matching engine first filters opcode sequences outputted from JavaScript de-obfuscation engine. For the opcode sequences not filtered out, JShield further matches them with detection format of opcode vulnerability signatures.

CHAPTER 3

Allocating Contents into Principals

In this chapter, I will first introduce configurable origin policy, which allocate contents from different origins into different principals and label those principals. Then, I introduce PathCutter, an approach isolate server-side contents to prevent worm propagation.

3.1. A General Framework of Allocating Contents (Configurable Origin Policy)

Web browsers have traditionally used the same-origin policy (SOP) to define protection boundaries between different web applications. According to SOP, a web site origin in the form of <*scheme, host, port*> serves as a label for the browser's security principals (isolated security domains). Each origin is protected from other origins in terms of resource access and usage.

With the advent of Web 2.0, modern web sites place new demands on browser's security that SOP was never designed to handle. Indeed, while intuitively simple, using web site origins to label browser principals has its limitations. Sometimes, SOP is too fine-grained. For example, contents from different web site origins (such as Gmail and Google Docs) may require unrestricted access to each other's resources, but SOP prevents browsers from rendering them as one principal. Other times, SOP is too coarse-grained. For example, it does not let browsers isolate logically different instances of web applications hosted on the same server, i.e., when one site hosts many distinct mashups, blogs, or user profiles, and it does not enable sites such as an email provider to run multiple, isolated sessions of the application in the same browser.

Faced with inflexibility of same-origin policy, web developers have worked around its limitations with a multitude of ad-hoc approaches. For example, subdomains may communicate by setting *document.domain* to a common suffix, a practice prone to security problems [**139**]. MashupOS [**151**] proposes a sandbox tag together with a verifiable-origin policy to isolate content from the same web site origin, which is particularly useful for mashups. Various cross-origin communication protocols are proposed [**46, 50, 63**] to break SOP for AJAX's XMLHttpRequest. Overall, however, such a piecemeal approach of working around SOP for specific resources or scenarios is prone to inconsistencies and security problems [**139**]. As shown in Section 3.1.1.2, the inconsistency leads to origin spoofing attacks in both cross-origin resource sharing (CORS) and new *iframe* tags of HTML5. Attackers can also escalate privilege in accessing *localStorage* of merged principals.

In this paper, we study a new way to label browser security principals. We propose a Configurable Origin Policy (COP), in which a browser's security principal is defined by a configurable ID specified by client browsers rather than a fixed triple <*scheme, host, port*>. Drawing inspiration from resource containers [**61**], we let the applications themselves manage their definition of an origin. That is, COP allows server-side and client-side code of a web application to create, join, destroy, and communicate with its own principals. In our scheme, one browser security principal can involve multiple traditional (SOP) web site origins, and various content from one traditional web site origin may render as multiple different principals. Fundamentally, COP origins break the longstanding dependence of web client security on domain names of servers hosting web content, and offer several compelling advantages:

- *Flexibility.* COP-enabled web applications can specify exactly which content from different domains can interact with one another on the client web browser. For example, Google may wish to let *gmail.com* and *docs.google.com* access each other's resources on the client. Moreover, with

COP, we can disable many ad-hoc, error-prone and potentially incoherent workarounds for SOP limitations [**139**], such as subdomain communication via *document.domain*, while still allowing sites to function correctly. COP also supports many scenarios that required a separate security mechanism, such as sandboxing mashups [**151**], and those that are not well supported by existing browsers, such as allowing a site to run different isolated web sessions in the same browser, all under one uniform framework.

- *Consistency.* In Section 3.1.1.2, we show that although all existing additions to SOP can merge or split an SOP-based browser principal, the newly generated principal has an inconsistent label with its security property.

  Configurable Origin Framework (COF, referring to modifications necessary to support COP on web clients and servers) is a unified framework that defines a new security label and mitigates all the inconsistencies.

- *Compatibility.* Because we change the web's central security policy, we undoubtedly face the challenges of deployment and backward compatibility. To address compatibility, we design SOP to be a special case of our new origin protocol, which makes COP compatible and secure with legacy web sites. COP-enabled web sites also remain compatible with existing browsers, since we convey our new design in existing protocols.

- *Lightweight-ness.* Our modification of WebKit to support COP is lightweight (327 lines), and our evaluation shows it has little overhead (less than 3%). Our modifications on web servers are also small. In examples we studied, less than ten lines were required to be inserted into existing server-side programs. To ease deployment, we also built a proxy that simulates modifications to web application code, and evaluated COP.

- *Security.* SOP is known to be vulnerable to attacks like cross-site request forgery [**9**], and recently-discovered cross-origin CSS attacks [**100**]. COP mitigates them.

	SOP	SOP+Additions	Other Non-SOPs	COP
Flexibility	No	Yes	Partial	Yes
Consistency	Yes	No	No	Yes
Compatibility	Yes	Yes	Yes	Yes
Lightweight-ness	Yes	No	Yes	Yes
Security	Low	Medium	Low	High

Table 3.1. Comparing COP with Existing Approaches.

For COP specific attacks, we show that even if an attacker sniffs the originID through an open HTTP connection, he cannot make additional damages through a COP design. Furthermore, we perform a *formal security analysis* by adopting and modifying an Alloy [**106**] web security model proposed and written by Akhawe et al [**57**]. The session integrity is ensured given the scope and the attack model.

3.1.1. Motivation and Related Work

The work is motivated in this section. We compare COP with SOP, SOP plus all the additions, and other non-SOPs. A high-level comparison is shown in Table 3.1.

3.1.1.1. Same-Origin Policy. The same-origin policy (SOP) is an access control policy defined in a client web browser, which allows only resources from the same <*scheme, host, port*> origin to access each other. Although SOP has been a good model with well-understood security properties, it has been inflexible for many modern Web 2.0 applications in the following two main scenarios.

Lack of Principal Cooperation. SOP makes it very difficult for multiple domains to be "combined" into one single principal. (i) Several domains, like `mail.google.com` and `docs.google.com`, may be controlled by a single owner who may want to allow sharing among the domains the owner controls. (ii) Since AJAX requests obey SOP, web applications cannot retrieve a URL from a different origin via XMLHttpRequest.

Lack of Principal Isolation. SOP makes it very difficult for one domain to be "split" into different principals. (i) A web site may require multiple isolated web sessions in one client browser. For example, a user may want to log in to multiple email accounts on the same provider site. (ii) To enrich users' experiences, many web sites embed third-party gadgets in iframes, such as Google Gadgets, Microsoft Widgets, and Facebook applications.

3.1.1.2. Existing Additions to Same-Origin Policy. To overcome such SOP inflexibility, web sites resort to a multitude of different approach to "patch" SOP.

- *Splitting One Single Principal.* Various approaches [**34, 95, 86, 82, 89**] create a separated environment to isolate different web sessions or third-party content at one client. New *iframe* tag in HTML5 [**20**] also supports a *sandbox* property to prevent access from the same origin.
- *Combining Multiple Principals at Server Side.* Many proposals [**46, 50, 63**] have been made to support cross-origin resource sharing (CORS). To support more fine-grained server-side merging, Content Security Policies (CSP) [**8**] and CSP-like mechanisms, such as SOMA [**129**], specifies access control policies at server side through HTTP headers or manifest files. Client browsers will be modified to enforce those specified policies.
- *Combining Multiple Principals at Client Side.* Since *document.domain* is insecure [**139**], Gazelle [**153**] disables changes to *document.domain* and proposes that sites use these existing cross-principal communication channels, such as *postMessage*. Other work, such as Object Views [**124**], layers complexity on top of cross-principal communication channels to facilitate easier object sharing.

Existing solutions to those problems are exclusive. For example, content security policies cannot substitute Object Views because CSP is specified at server side. However, in applications like Facebook Connect [**92**] and Google Friend Connect [**19**], cross-frame accesses are allowed dynamically after browsers receive server-side content.

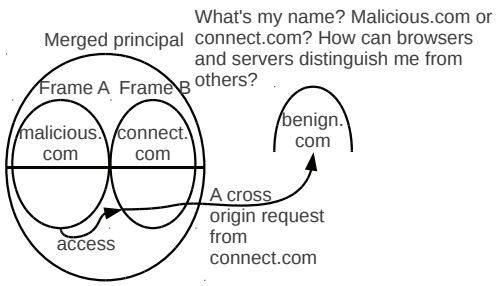

Figure 3.1. Origin Spoofing Attack for CORS (A has full access to B through a JavaScript library over postMessage channel. But this new combined principal can still send requests as B. What if C - benign.com, trusts B but not A?).

Consistency - COP vs. Existing Additions. Fundamentally, all these approaches fix flexibility problems in SOP, and thus they can split or merge different SOP principals. However, those newly created principals do not own a security label like the fixed triple (<*scheme, host, port*>) in traditional SOP principals. When the origin of new principals need to be inspected somewhere in those additions to SOP, browsers and servers can only check the old SOP origin label (<*scheme, host, port*>), which we call it mismatch between the principal and its security label (origin). Given the mismatch, a malicious principal can camouflage its identity and fool outsiders using SOP origin, which is defined as **origin spoofing attacks** in this paper. COP defines a new origin and resolves the mismatch.

Mismatch between a merged principal and its origin. For example, an isolated frame A from `malicious.com` is merged with another frame B from `connect.com` by Object Views [**124**] or a similar approach built on top of *postMessage* to achieve full transparent object access. The merged principal AB represents both `malicious.com` and `connect.com`. However, AB does not have a new origin to represent its new property.

- *Origin spoofing in cross-origin AJAX:* accepting requests from malicious principals. For a cross-origin AJAX request, servers will check the SOP origin in the *origin* header to decide the response. However, as shown in Figure 3.1, in the aforementioned merged principal case, A can

ask B to send a cross-origin request with the origin `connect.com` to a third party server, say `benign.com`. `Benign.com` cannot recognize the request is actually from a merged principal AB consisted of both `malicious.com` and `connect.com`.

- *Performance degradation in merging more than two principals:* redundant origin inspection. For example, after a frame from Facebook merged with a frame from Yelp, suppose the merged frames further want to merge with a frame from Twitter. In today's SOP model, because there is no new origin label introduced, the frame from `twitter.com` needs to check and merge with both frames separately with two *postMessage* channels. This becomes a serious scalability problem as the number of sites that want to communicate grows.
- *Privilege escalation in accessing localStorage:* incomplete origin inspection for shared objects. LocalStorage defined in HTML5 is shared across an SOP origin. Suppose frame A from `facebook.com` is merged with frame B from `yelp.com` by Object Views [**124**] or a similar approach built on top of *postMessage* to achieve full transparent object access. In this case, B can also access localStorage of the entire `facebook.com` domain, although B just wants to have full access to the specific frame, A.

Mismatch between a separated principal and its origin. For example, as supported by the new *iframe* feature in HTML5 [**20**], `a.com` can be separated into a gadget $G1$ (`a.com/benign`) and another gadget $G2$ (`a.com/malicious`) by specifying a *sandbox* property. However, both $G1$ and $G2$ do not have the same security property as `a.com`. When the browser or the server inspects principal's SOP origin for the purpose of security, either $G1$ or $G2$ can disguise itself.

- *Origin spoofing in* postMessage *channel:* navigating to malicious gadgets. The original *postMessage* design is vulnerable to a reply attack proposed by Barth et al [**66**]. Three frames, I (top), A (middle) and $G1$ (bottom), are nested. After $G1$ sends a message to I, the attacker A can

navigate its child frame $G1$ to a malicious one before $G1$ gets the reply. Therefore, browsers will check SOP origin of target frame in *postMessage*.

However, *the reply attack still exists* if the attacker navigates the benign gadget $G1$ to a malicious one $G2$ in the same SOP origin posted by the attacker. The browser only checks the SOP origin of $G2$ that is the same as $G1$ and thus the attack succeeds.

Lightweight-ness - COP vs. Existing Additions. In order to have the same flexibility as COP, all the additions have to be deployed upon current browser. The overhead is accumulated together. In particular, library built upon *postMessage* is extremely heavyweight.

For library built upon *postMessage*, such as Object Views [**124**], extra layer needs to be added. Objects are serialized into JSON representation, transmitted through *postMessage* channel, and then de-serialized back to objects. Therefore, object accesses in Object Views are more than 2X slower than native DOM accesses [**124**] due to object serialization and de-serialization even with native JSON support. In contrast, COP will enable direct object access between mutually trusting sites and suffer no performance penalty.

Security - COP vs. Existing Additions. *Document.domain* disobeys the least-privilege principle and can be insecure, as shown by Singh et al. [**139**]. Instead of *document.domain*, as shown in Object View [**124**] and Gazelle [**153**], web sites can use libraries over *postMessage* to facilitate communication. However, the authentication process through *postMessage* is often not used correctly even on popular web sites from reputable vendors, including Facebook Connect [**92**] and Google Friend Connect [**19**], as shown by Hanna et al [**98**]. In COP, we *merge* mutually trusting sites into one principal, as shown in Section 3.1.2.3, and enable object access between them directly via the DOM—the most familiar model for web developers.

3.1.1.3. Non-SOP Origin. As discussed in Section 3.1.1.2, additions to SOP is fundamentally inconsistent with SOP. A new origin label needs to be defined for inspecting the originating principal

at many places of both client and server side. In this section, we introduce previous attempts of defining non-SOP origins.

Finer-grained Label - (SOP + Something). A simple way of defining a non-SOP origin is to use SOP + something. Current HTML5 specification [**20**] defines an origin as <scheme, host, port, optional extra data>. In history, there are several ways to define optional extra data:

- *Subdomain and URL - <scheme, host, port, path>.* The "Building of a Secure Web Browser" project [**103, 104**] labels their principal with the URL of a page (protocol, host name plus path). Tahoma [**85**] lets web services specify a manifest file, which contains a suite of different domains which belong to one web site.
- *Locked Same Origin Policy - <scheme, host, port, public key infrastructure>.* Karlof et al. [**111**] integrates the public key infrastructure model into SOP to deal with dynamic pharming attacks. A similar idea is also proposed by Reis et al. in a position paper [**134**].
- *ESCUDO - <scheme, host, port, ring>.* ESCUDO [**107**] design a web browser protection model based on vetted mandatory access-control principals. They introduce a concept ring in addition to the SOP triple.
- *Contego - <scheme, host, port, capability>.* Similar to ESCUDO, Contego [**120**] adds a parameter - capability in addition to the SOP triple.

Limitations. Labeling principals by finer-grained labels (<scheme, host, port, optional extra data>) has the following inflexibility.

- *Lack of Support for Merged Origins.* <scheme, host, port, optional extra data> cannot represent merged origins. For example, a finer-grained origin cannot represent a principal merged by frame A from `a.com` and frame B from `b.com` through *postMessage* channel.
- *Lack of Client-side Creation.* Host and port in SOP triple are defined by web servers. When client browsers need to create a new origin for an iframe, they have to send a request to the

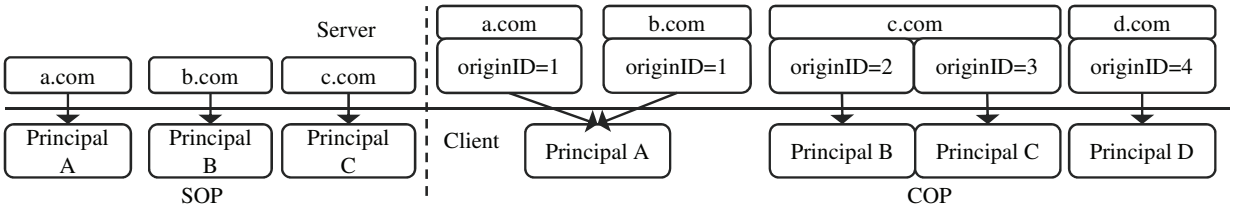

Figure 3.2. Content-to-Principal Mapping in SOP vs. in COP (originID is simplified for easy understanding).

server and wait for the round-trip delay. Furthermore, in offline mode that is often supported by many modern web applications, servers are not reachable to assign new origins.

Furthermore, locked same-origin policy is criticized by Jackson et al. [**105**] due to potential attacks, which they also find inherent in other finer-grained origin work. The locked SOP only considers supporting finer-grained origins but not collaboration of origins. It lacks an elegant way to let resources from other domains enter the existing origin.

Verifiable Origin Policy - Not a New Label. MashupOS [**151**] proposes a new origin policy called the verifiable origin policy (VOP). "A site may request information from any other site, and the responder can check the origin of the request to decide how to respond." This is a great proposal that is also adopted in COP.

However, regardless of name similarity, *VOP is orthogonal to either SOP or COP.* In both SOP and COP, a web site needs to check the origin (SOP origin or COP origin) of the request from the client browser. VOP does not define a new label (origin) but instead stress the fact that origin needs to be checked at server side. In particular, MashupOS cannot merge two principals at client side, like in Facebook Connect [**92**] case.

3.1.2. Design

3.1.2.1. Concepts in COP. Resources. Resources represent contents inside client side browsers and web servers. Examples of resources from the server are HTML files, images, script files,

etc. Examples of resources from the client are display, geolocation, and so on. Resources may be processed to generate further resources. For example, DOM is produced by rendering HTML files and modified by JavaScript code.

Principals. The concept of a principal, as borrowed from operating systems, in the context of web browsers is well discussed in previous work [**151, 153**]. It is an isolated security container of resources inside the client browser. Resources inside one principal are trusted by the principal itself. Resources outside principal X are not trusted by principal X but are trusted by the principal that the resources belong to. A principal is the atomic trustable unit in the browser.

We extend this concept in COP, where a principal is an abstract container that includes certain resources from both clients and servers with certain properties. A COP principal contains two parts, one on the server and the other on the client. The server-side's part of the COP principal is a worker, a thread or a process or a part of it, which serves the client. The client-side's part of the COP principal is what comprises a typical definition of a principal in a browser, an isolated container that is used to deal with contents from the server. For the rest of the paper, "principal" will refer to the COP principal in general.

Origins. An origin is defined as a label of a principal. Two principals that share the same origin will share everything between each other, which means they are essentially one principal. Two principals with different origins are isolated from each other. They can only communicate with each other through a well-protected channel.

OriginID. An originID is a **private randomly-generated identifier** used to annotate the origin of a principal. The originID is only known by the principal who owns it. Other principals cannot acquire the originID of principal X unless being told by principal X itself. In this sense, an originID is a *capability* to manipulate the principal it represents. OriginIDs are made arbitrarily hard to guess.

There are three reserved values of originIDs: *empty*, *default*, and *secret*. (i) The *empty* as a value of originID, specified by the client browser only, denotes a principal not associated with any content (hence the adjective *empty*). And the server will assign a value for the originID of such a principal. (ii) The *default* as a value of originID, denotes it is the same as the originID in current principal (both clients and server side included). (iii) The *secret* as a value of originID, denotes that the value of current principal's originID is not revealed by the owner.

Each resource in a principal will be labeled by an originID. With the originID, the client-side browser will decide in which principal to render the resource.

PublicID. A publicID provides a public identifier for a principal using which other principals can address this principal. It does not act as a capability like the originID to manipulate the principal it identifies. The publicID is designed to be publicly known. The browser maintains a table of correspondence of originIDs and publicIDs.

Principal's Server List (PSL). For each principal, the principal's server list (PSL), visible[1] to the users, is a list maintained by the browser to record all the servers or part of them that are involved in current principal by operations described later in Section 3.1.2.3. Each server in the list is represented in the format of <scheme, host, port, path>. For example, $http://www.a.com/part1$ denotes that all the resources and sub-directories under part1 of $http://www.a.com$ are participating in current principal. By default, in order to align with SOP, if not specified by the server, the default path will be $/$, denoting that the PSL includes the whole SOP origin.

3.1.2.2. Configurable Origin Policy (COP). With all these definitions, we can define our new origin policy, the configurable origin policy, as follows.

A principal (both server and client parts included) can configure any of its resources to an arbitrary but unique origin.

[1]One can adopt similar approaches in making the status of HTTPs certificate more noticeable by users for PSL.

This means that a principal can change its contents' origin to an arbitrary value. The program at the server side of the principal can configure the originID of the principal. For example, the server may send its content to clients together with an originID. The program at the client side can also configure the origin. For example, a client-side JavaScript program may change the originID of a document tree.

On server side, unlike the SOP model, in which the content-to-origin mapping is fixed for all contents from the same <scheme, host, port> tuple, in the COP model, we allow the principal to configure its own origin. An SOP origin can be split into several COP origins. As illustrated in Section 3.1.1.2, mashups and different web sessions are all examples. Similarly, multiple SOP origins can be combined together in configurable origins. For example, Google Docs and Gmail may want to share code or data dynamically, and thus they are better put in a single principal. Also, as illustrated in Section 3.1.1.2, `www.cnn.com` and `ads.cnn.com` can be combined into `cnn.com` origin. Figure 3.2 clearly shows the differences between SOP and COP content-to-principal mapping.

On client side, the principal is also given more freedom. In the classical SOP model, the switching of origins is not allowed at the client side. *document.domain* reduces this restriction only a little, which may not be enough for some applications, and that too at the cost of possible malicious access to the principal. For example, `a.com` and `b.com` cannot share the same principal, even when using *document.domain*. Because in the COP model, the origin ID of a principal is not tied to its location, the origin of a principal may be arbitrarily decided at the client side.

3.1.2.3. Operations on a COP Principal. We define the following operations on a COP principal.

Creating a Principal. A principal can be created by a server or a client by giving a new originID, as shown in Figure 3.3.

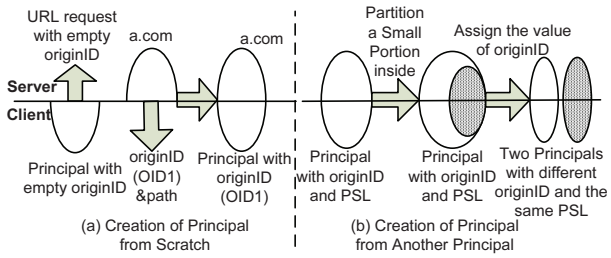

Figure 3.3. Creation of Principal.

Figure 3.3(a) illustrates the client requesting a URL together with an *empty* originID to *a.com* and the server sending the corresponding content with a new originID. In order to have multiple separate sessions from the same server, say for signing into multiple Gmail accounts, the server sends different originIDs to the client for different sessions. Given the different originIDs, the client browser renders the corresponding contents using different principals.

Clients can also create a principal as shown in Figure 3.3(b). A principal can assign a resource belonging to itself a new originID value to place this resource in a new principal. The child principal will inherit the PSL of its parent. Mashup isolation problem can also be solved using such client-side operations. Web integrators at client side can create different principals for content from different (distrusting) third parties by giving different originIDs based on the information provided by the server.

Joining an Existing Principal. Resources from one principal may wish to collaborate with, or *join*, resources from another principal. The joining process is discussed below.

As shown in Figure 3.4(a), a web site *a.com* may request to use a resource hosted on a different web site *b.com* under *a.com*'s principal — *a.com* can ask *b.com* to allow the resource to join *a.com*'s principal. Client browser supplies web site *a.com*'s originID and PSL when requesting that resource from *b.com*. If *b.com* agrees that this resource is allowed to join *a.com*, *b.com* will send the resource to *a.com* and attach *a.com*'s originID to it (case one in Figure 3.4(a)), and then client browser adds *b.com* plus the path specified by the server to the PSL of the current principal.

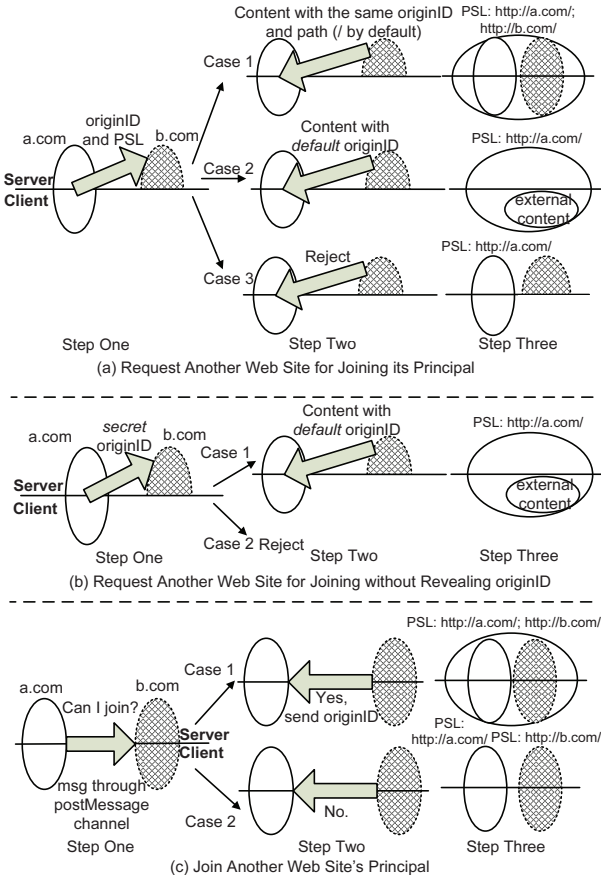

Figure 3.4. Joining another Principal (notice that one server acquires originIDs of other principals through clients not directly from other servers).

If *b.com* does not want to participate in the principal actively, it will send the resource back with *default* originID (case two in Figure 3.4(a)), and then client browser will not change the PSL of the current principal. If *b.com* refuses to let this resource join *a.com*, it will fail to respond with the resource (case three in Figure 3.4(a)).

This join operation can be used for *document.domain* problem. For example, when the client has a `www.cnn.com` principal and sends a request to `ads.cnn.com` with the principal's originID and PSL, `ads.cnn.com` will agree to join the existing principal with the same originID. On the other hand, a bank web site will generally not join an existing principal of another web site.

Second, as shown in Figure 3.4(b), a web site *a.com* may request to use a resource hosted on a different web site *b.com* under *a.com*'s principal without telling *b.com* its originID. Client browser supplies a *secret* originID when requesting that resource from *b.com*. If *b.com* agrees to provide that resource, it will send the resource with a *default* originID. Otherwise, *b.com* can reject that request the same as case three in Figure 3.4(a). In this case, *b.com* will not be participating in that principal but just provide external resource. In other words, *b.com* cannot control the principal.

This pseudo-join operation can be used for supplying cacheable contents. For example, *a.com* may request a cascading style sheet (CSS) by this operation since *a.com* does not want to reveal its originID to *b.com* and meantime *b.com* does not care which web site is using this style sheet.

Third, as shown in Figure 3.4(c), a resource or a principal from *a.com* may join another existing principal from *b.com*. The resource or principal from *a.com* acquires the originID of the other principal from *b.com* it wishes to join via an auxiliary communication channel (postMessage channel). By changing to this originID, the resource or the principal from *a.com* joins the other principal represented by this originID. And the PSL of the merged principal will also be the merging of those two principals' PSLs.

This case may be useful for collaboration among web sites. For example, Facebook Connect can be implemented with the join operation. A Facebook principal at the client browser may want to share information with another web site, say Yelp. The Facebook principal will create a new principal that is used for sharing and will then give the new originID to the other web site so that the other web site can join that newly created principal.

We have illustrated all possible join operations and corresponding cases. In summary, Figure 3.5 lists how a principal A and B may react based on the trust relationship between each other. Principal A initiates the join operation and Principal B receives the join request. A will trust,

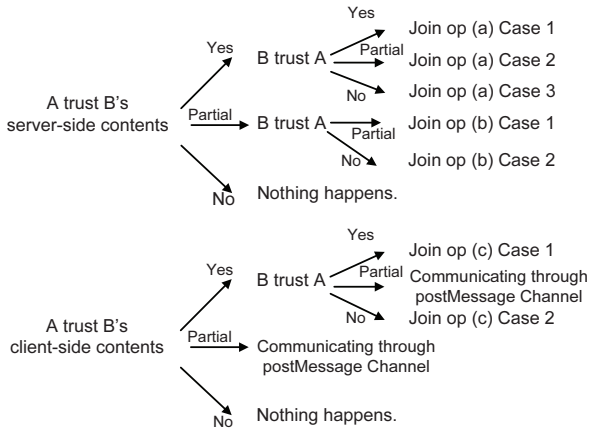

Figure 3.5. Making Different Decisions in Join Operation based on the Trust Relationship between Principal A (initiator) and B (receiver).

partially trust, or not trust B. "Partial" means that A trust part of B's resource but not all its behavior.

Communication inside a Principal. For client and server communication, accompanied by current originID, the communication with a server in PSL will be considered as a communication inside the current principal. The communication with a server not in PSL will ALWAYS considered as a join operation (therefore attached with originID and PSL).

Pure client-side communication inside a principal is unrestricted. Any resource can freely access any other resource. For instance, one JavaScript object can call the methods of another object.

Communication between Principals. Communication across principals can be achieved with explicitly defined and regulated channels. Namely, we can use the postMessage channel and the communication protocols for this channel [**66**].

Destroying a Principal. Principals may destroy themselves or be destroyed by a user. For example, a user may close all the tabs and windows belonging to a principal and in this way destroy it.

3.1.3. Implementation

Configurable Origin Framework (COF) requires both client-side and server-side modification. The server-side modification requires the server to send each resource together with the corresponding originID to the client. The client side needs to recognize the originID and put this resource into the corresponding principal.

Server-Side Modification. In our implementation, we modify the web application at server side so that resources in one web session will be allocated into one principal at client. This means the content-to-principal mapping is switched from SOP origin per principal to web session per principal. The concept of a session already exists in the present web, and denotes an information exchange process between the server and the client[**40**]. For example, when a user logs into his account on a web service, the web site sends a cookie to the user as an identity for further communication. Later on, any communication with that cookie is inside this session. The web server will check the identity cookie and reject requests without that cookie. In our paper, we adopt the existing concept of session. We will put resources of the same session[2] from the server into the same principal. Our server-side modification is discussed in Section 3.1.5.1. The rest of this section mainly deals with **client browser modification**.

Our client-side prototype implementation is based on WebKit [**47**], a popular open-source web browser framework. We demonstrate COF with the Qt browser that comes with WebKit. This browser uses WebKit's WebCore and JavaScriptCore. We insert 327 lines into WebKit to implement COF. The source code can be downloaded from configurable origin policy google code project [**18**].

[2]Notice that originIDs do not substitute session cookies, which still perform the same functionality as before.

In the rest of this section, we present originID and publicID generation, and WebKit COP enforcement respectively in Section 3.1.3.1 and 3.1.3.2. Then, we discuss the association of originID with different resources in Section 3.1.3.3 and 3.1.3.4. Finally, Section 3.1.3.5 presents compatibility issues.

3.1.3.1. OriginID and PublicID Generation. The representation of originID is similar to that of a session cookie: a long and random string. We generate a 128-bit random number by CSPRNGs (cryptographically secure pseudorandom number generator) [**10**] and encode it as the value of an originID.

For a principal X, a publicID is an identifier which can be used by other principals to refer to principal X. Once a principal is created, a unique publicID is assigned to the principal automatically by the browser. The browser maintains a table of publicIDs and its corresponding information, such as the domain name and the description from the principal itself. Other principals can use a new API `getpublicID(domain name)`, which returns a list of publicIDs belonging to the domain, to query this table for the information,.

3.1.3.2. Enforcing COP. Access control methods or other isolation mechanisms are required to protect the boundary of a principal. In COF, to put contents from the same COP origin into one principal we need to replace SOP-based access control mechanisms with those based on COP.

`SecurityOrigin` is defined as a class in the WebKit implementation for controlling access over all domains. It adopts SOP and Figure 3.6(a) shows its core function. In COF, we modify it to employ originIDs for access control. The key part of new design is shown in Figure 3.6(b). Resources labeled with the same originID belong to the same principal and can freely access each other. Since only small modifications are required for COP on WebKit, we believe it will be relatively easy to adopt COP for other browsers as well.

```
bool SecurityOrigin::canAccess(const
                        SecurityOrigin* other)const {
 ...
 if (m_protocol == other->m_protocol) {
  if (!m_domainWasSetInDOM
                     && !other->m_domainWasSetInDOM){
   if (m_host==other->m_host&&m_port == other->m_port)
    return true;
  } else if (m_domainWasSetInDOM
                      && other->m_domainWasSetInDOM){
    if (m_domain == other->m_domain)
     return true;
    }
 }
 return false;
}
```

(a) Access Control in SOP

```
bool SecurityOrigin::canAccess(const
                        SecurityOrigin* other) const {
  if (m_originID!=""  || other->originID()!="") {
        return m_originID == other->originID();
       }
    else {
    SOP Access Control
    }
}
```

(b) Access Control in COP

Figure 3.6. Access Control Implementation in SOP and COP.

HTTP Request	HTTP Response	HTML
HTTP/1.1 200 OK	GET /c.htm HTTP/1.1	<iframe originID=∗>
originID: *****	originID: ******	</iframe>
PSLPath:/part1	PSL: http://a.com/	
(a) OriginID and PSL with HTTP		(b) OriginID with HTML

Figure 3.7. Association of originID and PSL with Different Resources.

3.1.3.3. Association of OriginIDs with Resources. A principal is associated with a container for resources. We classify resources into two categories, resources from servers and dynamically-generated resources. Each resource belongs to one principal, implying that each resource may be associated with an originID.

Origins for Resources from Servers. Resources obtained from servers, such as HTML, images, and some of plugin data, are mostly transmitted via HTTP protocol[3]. As shown in Figure 3.7(a), we add a header, named originID, in the HTTP protocol to indicate the originID of the resource. When the browser sees this field, it will add this resource to the principal with this originID.

In addition, HTML can be used for originID association. For example, the content inside an *iframe* tag may belong to another principal but this cannot be represented via HTTP headers alone. As shown in Figure 3.7(b), for some HTML tags that will cause an HTTP request or can create a *document* object[4], we can have an originID different from the one of the main document. Since HTTP headers, HTML tags, and HTML attributes are designed to be extensible, our new modifications are completely compatible with existing browsers; they simply get discarded in existing browsers.

Some content, such as some plugin data, is not transmitted by HTTP protocol. Such content belongs to the principal which requested it. For example, a Flash program in a Flash plugin creates a TCP connection. Later, contents transmitted in this TCP connection will have the same origin as this Flash program. The Flash program belongs to dynamically-generated resources, which will be discussed in Section 3.1.3.3. In case the plugin program cannot be trusted, the whole plugin may be isolated in a different principal by assigning a different originID. We leave it as our future work to apply COP to plugin data.

Origins for Dynamically-Generated Resources. Dynamically generated resources refer to DOM, dynamic JavaScript objects, computed CSS, etc. These resources are derived from resources from servers. For example, DOM is generated from HTML parsing and JavaScript execution. There are

[3]HTTPS can be dealt with similarly because overall the confidentiality/integrity of the transfer channel problem is orthogonal to the problem of defining security principals, and can still use host names.

[4]Assigning a new originID is useful for only a few HTML tags, the ones that send another HTTP request or contains another DOM, such as *img* and *iframe*. For other HTML tags, because browser's access control is not fine-grained upon each DOM node, we cannot isolate them.

```
<script type="text/JavaScript">
//Inheritance--create an iframe with the same originID
(1) ifr1=document.createElement("iframe");
(2) document.getElementById("div1").appendChild(ifr1);
(3) ifr1.contentDocument.write("....");
//Dynamic Generation
// --create an iframe with a different originID
(4) ifr2=document.createElement("iframe");
(5) document.getElementById("div1").appendChild(ifr2);
(6) ifr2.contentDocument.write("....");
(7) ifr2.contentDocument.originID=generateOriginID();
</script> <div id="div1"> </div>
```

Figure 3.8. Origins For Generated Resources.

Table 3.2. Default Behaviors for HTTP Requests (Compatible with SOP).

HTTP Requests	Default Attached OriginID
Type a URL	*empty* originID from a new empty principal
HyperLink such as <a href=””>	URL in PSL: originID from the current principal URL not in PSL: *empty* value
Scripts or Stylesheet	*secret* originID
Embeded Object, like iframe and img tag	URL in PSL: originID from the current principal URL not in PSL: *empty* value
XMLHTTPRequest	URL in PSL: originID from the current principal URL not in PSL: *secret* originID

two types of policies for association of originID with these resources: inheritance and dynamic generation.

Inheritance is the default enforced policy. As shown in Figure 3.8, we create an iframe *ifr1*, which inherits the same originID from the HTML document (line 1). However, an originID can also be specified dynamically. As shown line 4 of Figure 3.8, the iframe *ifr2* is created and is given a unique but different originID value through *generateoriginID()* (line 7).

3.1.3.4. Transfer of Resources. Resources are transferred from the server to the client and across browser principals. In this section, we describe how COF secures client-server communications and how the browser mediates cross principal communications.

Client-server Communication - HTTP. As shown in Figure 3.7(a), the HTTP exchanges between the server and client are associated with an originID. As in the spirit of verifiable origin

policy [**151**], the originID of the request from a principal does not decide whether the corresponding response is accessible to the principal, and this principal is allowed to access the response only if the response carries the same originID as the principal's originID or *default* originID. (For *default* originID in the response, if the principal is empty, client browser will generate a new originID.) Now we discuss how the originID is used in the communication.

HTTP Request. HTTP requests in different operations have different behaviors. (i) Communication inside the current principal (a request to a server in PSL): launched from the current principal with its originID. (ii) Join operation (a request to a server NOT in PSL): launched from the current principal with its originID and PSL (iii) Create Operation (no matter whether the requested server is in PSL or not): launched from a different principal with that principal's originID.

To achieve those three requests, in COF, a principal can configure whether an HTTP request is from the current principal or a different one by specifying originID such as <img originID = "OID1">. However, to be convenient for programmers and compatible with SOP, the client browser can also attach an originID for those HTTP requests without originIDs specified explicitly, as shown in Table 3.2. The default policy aligns with SOP.

HTTP Response. An HTTP response is generated by the web server according to the HTTP request received. Based on different originIDs in the request and operations that the web server wants to perform, the web server will attach different originIDs in the response as listed in Table 3.3. For example, when the web server receives a request with a *empty* originID, it will send its response with a new originID and the client browser will adopt this originID as the originID for that empty principal.

An Example. Suppose a web page has an iframe <iframe originID= "OID1" src="example.com">. The browser will first create a principal with OID1 for the iframe. Then it will send a request with

Table 3.3. OriginID in HTTP Response According to Different HTTP Requests and Server's Decisions.

OriginID in HTTP Request	OriginID in HTTP Response			
	Join Operation	Comm inside Principal	Create Operation	Cacheable Content
empty	N/A	N/A	New Value	*default*
OID1	OID1	OID1	N/A	*default*
secret	N/A	N/A	N/A	*default*

OID1 to `example.com`. If `example.com` agrees to join the principal, it will send an HTTP response with header "originID: OID1". Therefore, the browser will render the response inside the OID1 principal. If `example.com` does not agree, it will send a 404 HTTP response or other error messages.

Communications between Principals. The *postMessage* channel facilitates cross-principal communication at client side. The usage of *postMessage2* is like *popup.postMessage2 ("hello!", popup.publicID);* due to the attack in Section 3.1.1.2. While *postMessage* takes an SOP origin as its second argument and performs an SOP check, *postMessage2* replaces this with a publicID check. The attack in Section 3.1.1.2 is mitigated because a malicious gadget always has a different publicID from a benign one.

3.1.3.5. Discussion on Compatibility. We present whether COP feature can be compatible with existing web servers, existing browsers and new HTML5 features.

Compatibility with Existing Servers. Existing servers don't specify an originID in their transmission. However, we can still be backward compatible with existing servers. We use a SOP tuple as an originID because SOP can be viewed as a special case of COP. We can still assign each SOP origin a principal if originID is not specified. Other COP-enabled principals are not allowed to switch their originID to any SOP tuple. At the same time, we need to allow *document.domain*. The security of older web sites neither improves nor worsens.

Compatibility with Existing Browsers. There are two possible options to make COP-enabled servers compatible with existing client browsers. First, existing servers can detect the client browser and deliver content accordingly, but this can be inconvenient. We have taken the second approach to convey originIDs in a new protocol field that older browsers will ignore. We have shown earlier in the section how this is accomplished for HTML and HTTP.

Compatibility with new HTML5 features. Some new features in HTML5, such as localStorage and FileSystem, are designed to grant access to a long-term identifier. Those features can be still supported in COP. Take localStorage for example. It can be modified to allow access from those principals with the same PSL. Therefore, a merged principal from both Yelp and Facebook cannot access the localStorage of pure Facebook.

3.1.4. Security Analysis

In this section, we first analyze possible attacks on COP and how such attacks can be mitigated. Then, we discuss whether COP can help defend against existing web attacks, such as CSRF. In the end, we perform a formal security analysis based on an existing web security model.

3.1.4.1. COP-Specific Attacks and Mitigation.

Leaking OriginIDs. OriginID is an essential and secret identity for a principal. We neet to prevent leaking originIDs.

Protecting OriginIDs. Given the similarity between originIDs and session cookies, methods of protecting session cookies can also be used for protecting originIDs.

- *Server-side protection:* Reusing protection mechanisms for session cookies. OriginIDs are generated *dynamically* and stored safely the same as session cookies on server side.
- *Protection during transmission:* HTTPS.

- *Client-side protection:* (i) Preventing originID access from a different principal through a sandbox approach [**146**] or JavaScript rewriting approaches [**132**]. (ii) Channel bound originID. Similar to channel-bound cookies [**1**], originID can be made channel-bound too. Even if an attacker acquires a channel-bound originID, he cannot authenticate it with the server via other connections.

How do we prevent originID leaks by a careless programmer's mistake? Two methods are adopted to prevent leaking originID by a mistake: (i) Defaulting safe behaviors. As shown in Table 3.2, default behaviors of sending originIDs are restricted within known servers (if no merging occurs, there is only SOP server). (ii) Using secret originID. In case that the programmer does not know how to use originID correctly, he can use *secret* originID to prevent originID leaks.

Given two aforementioned protection mechanisms, we believe a web site will seldom send its originID to a malicious source by, for example, including third-party content through iframes in a wrong way. It is the same as the fact that a web site rarely includes a malicious script directly or carelessly sends its session cookie to a malicious server.

What if originID is leaked through an HTTP connection? Let us discuss a scenario where a web site *benign.com* is using HTTP and originID can be sniffed. We have the following two arguments: (i) Overall small chance. (ii) Even if it happens, there is no additional damages brought by COP.

First, the chance that the contents of *benign.com* can be manipulated is small. Clients need to visit *benign.com* in an open network such as coffee shop. Then, the attacker needs to lure the client to visit *malicious.com* in the same browser in order to join and manipulate *benign.com*'s principal.

Secondly, even if those two conditions are satisfied, the damage an attacker could make is the same as what he could do from sniffing the network and luring people to visit *malicious.com*.

- Contents of *benign.com* can be directly sniffed or acquired by a sniffed session cookie. Meanwhile, the attacker can also make changes to user's contents on benign.com by the sniffed session cookie.
- If *benign.com* does not have session cookie, phishing by altering contents in a merged principal is not different from the one in pure *malicious.com*, because after merging, *malicious.com* is in PSL, which is easily visible to users in COF.

What is the security implication for Principal A to give its originID to Principal B in order to configure Principal A? It means that Principal A totally trusts Principal B. For example, ads.cnn.com completely trusts www.cnn.com, however, they currently use *document.domain = 'cnn.com'* to communicate with each other, which is error-prone, as shown by Singh et al [**139**]. In COP, A (ads.cnn.com) can give its originID to B (www.cnn.com).

Potential Attacks when Combining SOP and COP. Interaction of web content following COP and web content following SOP may lead to attacks. In a web integrator where all its isolated gadgets are from the same domain, an attacker might modify a principal with originID back to a principal with SOP origin by removing the originID, so that the attacker can access another principal with the same SOP origin but not the same COP origin. We resolve this problem by always using COP when either of two principals is using COP. The originID of a SOP site will be derived from the SOP triple and hence will be different from every COP originID. In this case, two principals need to use the *postMessage* channel to communicate with each other.

Another attack is to integrate COP web sites with SOP web sites. For example, an SOP web site is embedded inside a COP web site using an iframe. COF can deal with this case because SOP is a special case in COP. If we don't find originID specified, we will consider the <scheme, host, port> to be a special originID which is different from any other originID specified by COP web sites. COF will always put contents in the iframe from SOP sites into a separate principal. We can

always differentiate COP and SOP web sites because COP web sites will always have an originID HTTP header.

Principal Hijacking Attack. Given the similarity between a session cookie and an originID, the attacks to session cookies, such as session substitution/fixation [**72**], also need to be considered here. Translated to originID attack, a session substitution/fixation will be as follows.

An attacker M visits `benign.com` on his own and acquire the originID $OID1$ for his principal A. The attacker triggers the client to visit his own web page `malicious.com` and set the originID of the malicious principal B to $OID1$. The malicious principal B sends a request to `benign.com`. Then, `benign.com` will consider the $OID1$ to be within the attacker's principal A and return contents. Client user will see a web page from `benign.com` but controlled by M.

Defense. When the malicious principal B sends a request to `benign.com`, since `benign.com` is not in the PSL of B, the client browser will ask `benign.com` to join B with B's originID and PSL (`malicious.com`). `Benign.com` can recognize that B is controlled by `malicious.com` and thus decline the request.

3.1.4.2. Mitigating Existing Attacks. Cross-Origin CSS Attacks. Cross-Origin CSS attacks were described recently by Huang et al. [**100**]. The attacker may inject some crafted content into *example.com* using blog posts or wall posts and then use their site *attack.com* to import *example.com* as a CSS stylesheet. When a user visits *attack.com*, the confidential information of that user from *example.com* will be stolen. If COF were adopted here, because the server will check the originID of the principal that sends the request, *example.com* will reject the request, thus preventing *attack.com* from importing its contents as a stylesheet.

***Document.domain* Threat.** *Document.domain* threat is described by Singh et al [**139**]. For example, when a web page from *x.a.com* sets its domain to *a.com*, a web page from *y.a.com*, which is

compromised by the attacker, can access the resource of that web page of *x.a.com* by setting its domain to *a.com*. This disobeys least privilege, as access control is relaxed too broadly.

In COF, only web pages that know the originID belong to the same principal. For the *document.domain* example above, even if an attacker compromises *y.a.com*, he cannot access any resource from *x.a.com* in a COP principal because he doesn't know the originID at client side.

Cross-Site Request Forgery (CSRF). CSRF is an attack which forces execution of unwanted actions from an end user on a web application in which s/he is currently authenticated [**9**]. A typical example of CSRF is an img tag such as

<img src = "http://bank.com/transfer.do?acct=A&amount=1000" width = "1" height = "1" border = "0"> .

When embedded inside a web page, it triggers browsers to fetch this link, causing the server to execute the "transfer" action. As we can see, there are several steps in CSRF. First, the link needs to be embedded on the web site. Second, the browser needs to send the request. Third, the server (the bank in this case) needs to allow this action. Defenses involve mitigation at any of the above steps.

Barth et al. [**65**] have analyzed CSRF and defenses against it comprehensively. They propose the origin header, which is similar to the referrer header but without the path and query parameters so as to ensure user privacy.

In COF, the originID header can effectively play the role of the origin header. In step three, the web server will use the originID (sent in step two) to determine if the request originated from its own principal and thus avoid CSRF attacks by declining the action from a different principal.

Origin Spoofing Attacks. As discussed in Section 3.1.1.2, an origin spoofing attack is launched by a merged or separated principal using an old SOP origin to camouflage itself. In COF, we define

originID as the new principal's label, which can be checked by a client browser or a web server, thus mitigating origin spoofing attacks.

3.1.4.3. Formal Security Analysis. Background. Akhawe et al. [**57**] abstract a formal web security model and build the model based on Alloy, "a model finder: given a logical formula, it finds a model of the formula" [**106**]. They apply the model upon five web security mechanism (the Origin header, Cross-Origin Resource Sharing, Referer Validation, HTML5 forms, and WebAuth), and discover two known vulnerabilities and three unknown vulnerabilities. A recent paper [**82**] also adopts their model to find design flaws.

Modeling. We modify their model to switch same-origin policy to configurable origin policy. The $owner$ property of $ScriptContext$ points to a COP origin instead of an SOP origin. All the operations are introduced for COP origin. For example, a create operation is as follows.

```
pred createCOPOrigin[aResp: HTTPResponse]{
 one originID:COPOrigin |
 originID !in
           (univ.theReqCOPOrigin&univ.theRespCOPOrigin)
 implies {
  (aResp.headers&RespOriginIDHeader).theRespCOPOrigin=
                                                originID
 }
}
```

We check the COP origin in each HTTP response to let the response fit into different principals.

```
fact COPOriginMatch{
 all sc : ScriptContext, t:sc.transactions |
 sc.owner =
  (t.resp.headers&RespOriginIDHeader).theRespCOPOrigin
 or
  (t.resp.headers&RespOriginIDHeader).theRespCOPOrigin
```

```
                                    = defaultOriginID
}
```

Experiment Setup. The attack model that we are using is the *web attacker* model introduced by Akhawe et al. [**57**]. The attacker controls malicious web sites and clients but does not master the network. Therefore, he cannot sniff or alter the contents on the network. The Alloy codes of the attacker model are inherited from Akhawe et al[**57**].

```
check checkSessionIntegrity{
    no t:HTTPTransaction | {
      some t.resp
      some (WEBATTACKER.servers &  involvedServers[t])
    }
} for 5 but 0 ACTIVEATTACKER, 1 WEBATTACKER,
 1 COPAWARE, 0 GOOD, 0 SECURE, 0 Secret, 1 HTTPClient
```

Results. Alloy is not able to find any counterexample by operations that are not considered by COF, implying that the session integrity is ensured given the attack model and limited scope.

3.1.5. Evaluation

First, we will discuss the practicality of deploying web applications using COP in Section 3.1.5.1. Next, we will evaluate COF's performance in Section 4.1.5.3. Then we discuss the compatibility in Section 3.1.5.3. We use a client with a 2.5GHz CPU with 16GB memory, and a server with a dual-core 1.6GHz CPU and 1GB memory, running Apache-PHP-MySQL. Both of these machines are on the same local network.

3.1.5.1. Deploying Web Applications.

Migrating Existing Code. To fully deploy COF, we need browser support and server-side support. For server-side support, the server-side application code needs to be modified to use COF.

```
protected function _validate() {
  ...
  if (validation failed)  return false;
  if (checkPSL()) return false;
  if (isEmptyOriginID()) createOriginID();
  header('originID:'.getOriginID(session_id()));
  //get originID from sessionID-to-originID mapping
  return true;
}
```

Figure 3.9. Modification on Varien.php of Magento. Each originID is mapped to a session ID. Session ID still takes its role in authenticating users, while originID is used to differentiate and isolate principals.

By modifying several popular web applications, we demonstrate that such modifications are light-weight and easy perform.

Proxy Assistance. Because we don't have control over many web servers, we designed a COF server-side proxy that mediates communication between servers and clients. The COF proxy, which can be found at [**18**], adds COF support to unmodified web sites to demonstrate our idea.

CNN is using *document.domain* to merge two of its domains: `www.cnn.com` and `ads.cnn.com`. When we disallow *document.domain*, an advertisement iframe is missing because the JavaScript access between the main page and the iframe is denied. When deploying our proxy, and disallowing *document.domain* in COF, the CNN web site can still display its content correctly This demonstrates that COF can achieve site collaboration without using *document.domain*.

Server-side Modification. We show how to adopt COF upon server-side applications and demonstrate the relative ease of modifying server-side code. We take web applications with login sessions as an example. The login cookie or session ID assigned by the server is mapped to a unique originID. We can reuse the validation of session ID or login cookie as the validation of originID. We changed one popular web application − Magento, to demonstrate our approach.

Our example Magento [**122**] is a top eCommerce software platform used by more than 60,000 online stores. It is written in PHP and runs in the Apache, PHP and MySQL platform. Magento

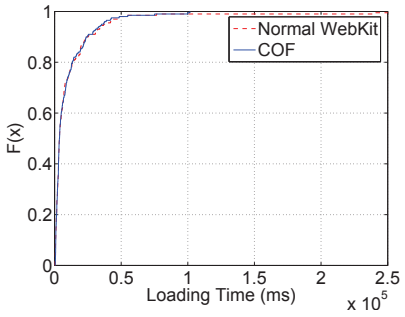

Figure 3.10. CDF of Loading Time with COF and with Normal WebKit.

adopts PHP built-in session management. As shown in Figure 3.9, we just need to generate a unique originID for each session ID.

Utilizing New Features in COP. As an example, we create a mini web integrator using COP features below.

There are isolated mashups from the same domain in our web integrator. We create different originIDs for different gadgets.

```
<?php
    function generateOriginID() { ...
    }
    header('originID:'.generateOriginID());
    if COPSupportedBrowser() { ?>
     <iframe src="..." originID=
         <?php echo generateOriginID();  ?> >
     </iframe>
     <iframe src="..." originID=
         <?php echo generateOriginID();  ?> >
     </iframe>
     ...
<?php } else {...} ?>
```

3.1.5.2. Performance Evaluation. The loading time of web pages under COP is measured with WebKit modified to support COF and with a COF proxy. The loading time of web pages under SOP is measured with unmodified WebKit. We use the time when the request is made as the starting time of loading a web page and the time of firing of the JavaScript *onload* event as the end time of loading a web page. Alexa top 200 web sites [**3**] are evaluated.

Figure 3.10 shows the results. We compare the cumulative distribution function (CDF) of loading time under COP to the one under SOP. The curve is almost the same which means COF brings little delay. The results are not surprising because little time is spent in *SecurityOrigin* checks when compared to other tasks like rendering, parsing, and JavaScript execution.

3.1.5.3. Compatibility Evaluation. We use Alexa top 100 web sites and visually compare the sites rendered with a COP-enabled browser and with an unmodified browser. For some web pages that require login (like Facebook), we log in first. We also follow some of the links on the web page to explore the functionality of that web page. For example, we search with some keywords on Google. We interact with many web sites like Facebook, e.g., by clicking menus, posting messages on friends' wall, and looking at profiles of other people. As expected, all the 100 web sites show no difference when rendered with a COP-enabled browser and when rendered with an unmodified browser.

3.2. A Case Study (PathCutter)

JavaScript is a cornerstone of the modern Internet that enables enhanced user interactivity and dynamicism. It is universally adopted by all modern e-commerce sites, web portals, blogs, and social networks. However, JavaScript code also has a demonstrated penchant for attracting vulnerabilities. JavaScript-based Cross Site Scripting (XSS) worms pose a severe security concern to operators of modern social networks. For example, within just 20 hours in October 4, 2005, the

MySpace Samy worm [**31**] infected more than one million users on the Internet. More recently, similar worms [**53, 54, 55**] have affected major social networks, such as Renren and Facebook, drawing significant attention from the public and media.

JavaScript worms typically exploit XSS vulnerabilities in the form of a traditional XSS, Document Object Model (DOM)-based XSS, or content sniffing XSS vulnerabilities. Incorporated into web applications, Javascript worms can spread themselves across social networks. Although they are referred to as worms, these JavaScript malware activities are more akin to viruses, in that they rely on interactions by users on the social network to replicate themselves. Once a vulnerable user is infected, malicious logic residing on the user's page coerces browsers of other victim visitors to replicate the malicious logic onto their respective pages.

The high degree of connectivity and dynamicism observed in modern social networks enables worms to spread quickly by making unsolicited transformations to millions of pages. While the impact of prior XSS worms has been quite benign, it is conceivable that future worms would have more serious implications as underground economies operated by cybercriminals have become increasingly organized, sophisticated, and lucrative. In [**99**] Billy Hoffman describes a hypothetical 1929 Worm that uses a self-propagating XSS attack on a brokerage site to wreak havoc on financial markets.

The growing threat of XSS worms has been recognized by the academic community, notably in the following two papers. The Spectator [**116**] system proposed one of the first methods to defend against JavaScript worms. Their proxy system tracks the propagation graphs of activity on a website and fires an outbreak alarm when propagation chains exceed a certain length. A fundamental limitation of the Spectator approach is that it does not prevent the attack propagation until the worm has infected a large number of users. In contrast, Sun et al. [**143**] propose a purely client-side solution, implemented as a Firefox plug-in, to detect the propagation of the payload of

a JavaScript worm. They use a string comparison approach to detect instances where downloaded scripts closely resemble outgoing HTTP requests. However, this approach is vulnerable to simple polymorphic attacks.

In this paper, we propose PathCutter as a complementary approach to XSS worm detection that addresses some of the limitations of existing systems. In particular, PathCutter aims to block the propagation of an XSS worm early and seeks to do so in an exploit agnostic manner. To achieve its objectives, PathCutter proposes two integral mechanisms: *view separation* and *request authentication*. PathCutter works by dividing a web application into different views, and then isolating the different views at the client side. PathCutter separates a page into views if it identifies the page as containing an HTTP request that modifies server content, e.g., a comment or blog post. If the request is from a view that has no right to perform a specific action, the request is denied. To enforce DOM isolation across views within the client, PathCutter encapsulates content inside each view within pseudodomains as shown in Section 4.1.3. However, isolation by itself does not provide sufficient protection against all XSS attacks. To further prevent Same Origin Cross Site Request Forgery (SO CSRF) attacks, where one view forges an HTTP request from another view from the same site, PathCutter implements techniques such as per-url session tokens and referrer-based view validation to ensure that requests can be made only by views with the corresponding capability.

The design of PathCutter is flexible enough to be implemented either as a server-side modification or as a proxy application. To evaluate the feasibility of a server-side deployment, we implement PathCutter on two popular social web applications: Elgg and WordPress. We find that only 43 lines of code are required to inject PathCutter protection logic into WordPress and just two lines[5] of additional code together with an additional file of 23 lines of code are required to

[5]Elgg has built-in support for request authentication but not view separation.

secure Elgg. We also evaluate a proxy-side implementation of PathCutter. The proxy seamlessly modifies content from popular social networks like Facebook on the fly to provide protection from XSS injection attacks.

Based on published source code and press reports, we analytically investigate PathCutter's efficacy against five real-world JavaScript worms: Boonana [**54**], Samy [**31**], Renren [**53**], Space-Flash [**42**], and the Yamanner worm [**26**]. Together, these worms span diverse social networks and exploit various types of XSS vulnerabilities, including Flash XSS, Java XSS, and traditional XSS. However, they converge in their requirement to send an unauthorized request to the server in order to spread themselves. PathCutter exploits this need to successfully thwart the propagation of all these worms. Finally, we conduct performance evaluations to measure the overhead introduced by our PathCutter implementation at the client side. Our results show the rendering overhead (latency) introduced by PathCutter to be less than 4% and the memory overhead introduced by one additional view to be less than 1%. For highly complex pages, with as many as 45 views, the additional memory overhead introduced by PathCutter is around 30%.

Contributions: Our paper makes the following contributions in defending against XSS JavaScript worms:

- We identify two key design principles (view separation by pseudodomain encapsulation and request authentication) for fortifying web pages from XSS worms.
- We develop prototype implementations of the server-side and proxy-side designs.
- We validate the implementation against five real-world XSS social network worms and experimental worms on WordPress and Elgg.
- We demonstrate that the rendering and memory overhead introduced by PathCutter is acceptable.

3.2.1. Problem Definition

A cross-site scripting (XSS) attack refers to the exploitation of a web application vulnerability that enables an attacker to inject client-side scripts into web pages owned by other users [**52**]. To illustrate how PathCutter blocks the propagation of a JavaScript-based XSS worm, we begin by describing the steps involved in the life cycle of a typical XSS worm exploit. Although XSS worms exploit different types of XSS attacks, they all share a need to acquire the victim's privilege (in Step 2) and thus issue an unauthenticated cross-view request (in Step 3), which PathCutter seeks to block.

: *Step 1 – Enticement and Exploitation:* A benign user is tricked into visiting (or stumbles upon) a malicious social network page with embedded worm logic that has been posted by an attacker. The worm is in the form of potentially obfuscated, self-propagating JavaScript, which is injected via an XSS vulnerability.

: *Step 2 – Privilege Escalation:* The malicious JavaScript exploits the XSS vulnerability to gain all the victim's rights and privileges to all websites that are currently connected to from within the victim's compromised browser. For example, if the victim is logged into a social network account, the worm has the ability to modify the victim's home page and can send messages to the victim's friends.

: *Step 3 – Replication:* The worm now replicates itself. As shown in Figure 3.12(a), the JavaScript worm uses its victim's own privileges to send the social network web server a request to change the victim's home page. The victim's home page is now altered to include a copy of the Javascript worm.

Table 3.4. Design Space of XSS Worm Defense Techniques

	Spectator [116]	Sun et al. [143]	Xu et al. [159]	BluePrint [149]	Plug-in Patches	Barth et al. [62]	Saxena et al. [137]	PathCutter
Blocking Step	4	3	4	1	1	1	1	2
Polymorphic Worm Prevention	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
Early-Stage Prevention	No	Yes	No	Yes	Yes	Yes	Yes	Yes
Types of XSS JavaScript Worms that Can Be Defended	All	All	Passively Observable Worms	Traditional Server-Side XSS Worms	Plug-in XSS Worms	Content Sniffing XSS Worms	DOM-based XSS Worms	All
Deployment	Server or Proxy	Client	Server	Server	Client	Client	Client	Server or Proxy
Passive/Active Monitoring	Active	Passive	Passive	Active	Active	Active	Active	Active

: *Step 4 – Propagation:* When other benign users subsequently visit the infected victim's page, Steps 2 and 3 are repeated. Such a strategy has been demonstrated in the wild to support worm epidemics that can grow beyond a million infections.

3.2.2. Related Work

3.2.2.1. XSS and JavaScript Worm Defense. Researchers have proposed numerous defenses against XSS attacks and JavaScript worms that directly relate to our work. A comparison of our work with closely related work is shown in Table 3.4. Each individual defense mechanism targets different stages of an XSS worm propagation and it can be deployed at either the client or server. We explore different XSS attack strategies and defenses in more detail below.

Cross-Site Scripting Attacks and Defenses: Cross-site scripting attacks can be broadly classified into two categories, traditional *server-side* XSS attacks (stored and reflected [**52**]), and *client-side* XSS attacks (DOM-based XSS [**12**], plug-in-based XSS [**16**], and content sniffing XSS attacks [**62**]), as shown in Figure 3.11.

In a *traditional XSS attack*, clients receive injected scripts from the server. Many techniques have been proposed that operate at the server side to defeat traditional XSS attacks, including [**83,**

157, 110, 59, 70, 101, 117, 123]. While these systems are quite successful at identifying XSS vulnerabilites, their tracking of information flow is restricted to the server side and is blind to the client-side behavior of browsers and vulnerabilities in browser plug-ins. BEEP [**109**] and Noxes [**113**] are the first client-side systems to defend traditional server-side XSS attacks. Later, recent papers on systems such as Blueprint [**149**] and DSI [**128**] discuss browser quirks [**51**] and propose client-side solutions to traditional XSS attacks. Bates et al. [**68**] criticize client-side filtering and propose their own solutions. Content Security Policy (CSP) [**8**], proposed by Mozilla, injects a very fine grained policy that is specified at the sever side into HTTP headers. This policy is then adopted by client browsers and enforced during every HTTP request.

In a *DOM-based XSS attack*, clients inject scripts through an unsanitized parameter of dangerous DOM operation, such as $document.write$ and $eval$. A simple example is that of the client-side JavaScript of web application calls $document.write(str)$ where str is part of $window.location$. Therefore, the attacker can inject scripts into parameters of URLs. A few defense mechanisms [**137**] are proposed for DOM-based XSS. Futhermore, CSP can be used to prohibit the use of dangerous functions such as *eval* and *document.write*, but such policies also limit website functionality. Recently, Barth et al. [**62**] proposed a new class of XSS attack called *Content Sniffing XSS attacks* where an image or a pdf file may also be interpreted as a JavaScript file by the client browser. Moreover, malicious JavaScript could also be injected by plug-ins. This has led to the proliferation of plug-in-based XSS vectors such as *Flash-based XSS attacks*, as a means to inject scripts into web pages. For example, the Renren worm [**53**] exploited a Flash vulnerability to enable access to the infected web page vulnerability, and inject malicious JavaScript. To fix the attack, users had to update their Adobe Flash plug-in to prevent such malicious accesses. These defenses all target **Step 1** in the propagation of an XSS worm.

JavaScript Worm Defense Techniques: Sun et al. [**143**] propose a Firefox plug-in that detects JavaScript worms using payload signatures. Their mitigation mechanism targets **Step 3** (Replication) in the life cycle of an XSS worm. Their approach is limited in that it protects only the specific client and not the entire web application. Furthermore, it is vulnerable to polymorphic worms where the payload dynamically changes during the worm's propagation. As shown by Dabirsiaghi et al.[**88**], the next generation of JavaScript XSS worms could integrate advanced polymorphic payloads, which may prevent direct payload fingerprinting of worm instances and thereby prolong and widen the epidemic.

Spectator [**116**] adopts a distributed tainting and tagging approach that detects the spreading behavior of JavaScript worms. The mitigation mechanism, which can be implemented as a proxy, targets Step 4 (Propagation) in Section 3.2.3. A deficiency of their approach is that it only detects the worm once a critical mass of users is been infected. Xu et al. [**159**] propose building a surveillence network that uses decoy nodes to passively monitor the social graph for suspicious activities. Their approach is complementary to ours in that it can detect worms like Koobface, that spread through malicious executables and are delivered through remote browser exploits, which PathCutter cannot. In contrast, they acknowledge that their approach cannot detect worms like MySpace Samy because it "does not generate passively noticeable worm activities". Another limitation of their approach is that Xu's decoy nodes, like Spectator, require a minimal threshold of users to be infected before detection. Both of these graph-monitoring systems target **Step 4** in the propagation of an XSS worm.

3.2.2.2. Request Authentication and View Separation Techniques. Two main techniques used in PathCutter include request/action authentication and view separation. Here, we discuss related

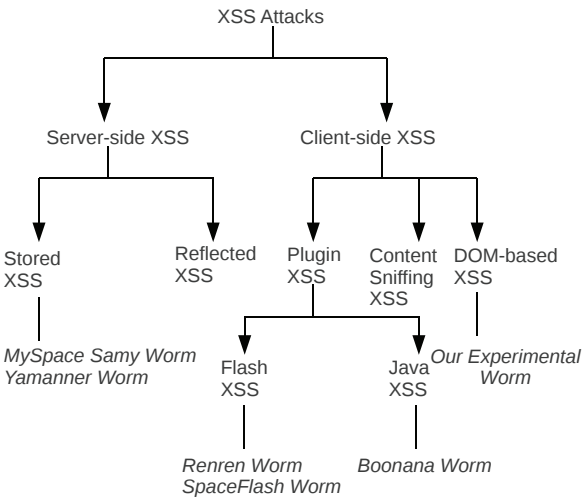

Figure 3.11. Taxonomy of XSS JavaScript Attacks and Worms

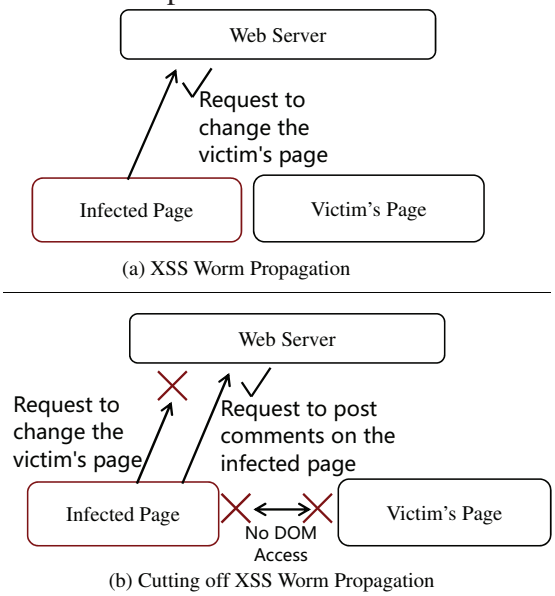

Figure 3.12. XSS Worm Propagation

work that informed the development of these two techniques.

Request/Action Authentication Techniques: Barth et al. [**65**] proposes the use of an *origin* HTTP header to validate the origin (<scheme, host, port>) of an HTTP request. Previous attempts have also used secret tokens and the *referer* header to validate requests from a different origin.

Defending against CSRF attacks is similar to cutting off the propagation path of a JavaScript worm in the sense that both of them need to validate the request. Therefore, *referer* and secret tokens can be used in both cases. However, there is also the following fundamental difference. A CSRF attack is across different SOP origins but a JavaScript XSS worm propagation usually happens within the same SOP origin. For example, `malicious.com` may try to modify contents on `bank.com`. Those two web sites are properly isolated at the client side. Compared to a typical CSRF attack, a JavaScript worm spreading is much harder to defend and detect, because the forged request is actually from the same web site—the MySpace worm spreads within MySpace, so SOP is not violated). And in theory, the worm can modify the user's contents from the client side because they are in the same origin. Hence, although we leverage CSRF defense methods within PathCutter, they cannot by themselves prevent XSS worm propagation as the *origin* header proposed by Barth et al. [**63**] contains only origin information such as `http://www.foo.com` and cannot distinguish requests from the same origin. To better illustrate this point, secret tokens are indeed adopted by MySpace. However, because the token is accessible from the same origin, the MySpace Samy worm [**31**] successfully steals that token. Similarly, attackers can also easily create and control an iframe with the same origin URL to make a request with correct *referer* header.

As a complementary strategy, social networks could also incorporate techniques such as CAPTCHAs [**5**] to authenticate actions/requests and defend against XSS attacks, at the cost of some user inconvenience. Similar defense techniques are used by websites such as Amazon that always prompt users to input their username and password before performing a potentially dangerous action.

View Separation Techniques: MiMoSA [**60**] proposes a view/module extraction technique to detect multi-step attacks at server side. Their concept of a view is different from ours, and they can detect only traditional server-side XSS vulnerabilities. Many blogs such as WordPress adopt

different subdomain names like `name.blog.com` to augment users' self-satisfaction of owning a subdomain. Since the purpose is not actually to prevent XSS worms, they do not really combine view separation with request authentication. View separation is also very coarse, such that vulnerable actions cannot always be isolated. For example, in many social web networks, such as Facebook, updates from your friends will also be shown on your own page, thus launching unauthorized requests. In PathCutter, contents in the same page can be separated into different views. Finally, there has been a recent research thrust [**96, 85, 153, 133, 145, 146**] on building better sandboxing mechanisms for browsers. We argue that these approaches are complementary. While sandboxing provides a strong containment system, it is entirely up to the programmer to decide which contents to put into the container. In PathCutter, we can adopt any of these approaches to make the isolation of different views stronger.

3.2.3. Problem Definition

A cross-site scripting (XSS) attack refers to the exploitation of a web application vulnerability that enables an attacker to inject client-side scripts into web pages owned by other users [**52**]. To illustrate how PathCutter blocks the propagation of a JavaScript-based XSS worm, we begin by describing the steps involved in the life cycle of a typical XSS worm exploit. Although XSS worms exploit different types of XSS attacks, they all share a need to acquire the victim's privilege (in Step 2) and thus issue an unauthenticated cross-view request (in Step 3), which PathCutter seeks to block.

: *Step 1 – Enticement and Exploitation:* A benign user is tricked into visiting (or stumbles upon) a malicious social network page with embedded worm logic that has been posted by an attacker. The worm is in the form of potentially obfuscated, self-propagating JavaScript, which is injected via an XSS vulnerability.

: *Step 2 – Privilege Escalation:* The malicious JavaScript exploits the XSS vulnerability to gain all the victim's rights and privileges to all websites that are currently connected to from within the victim's compromised browser. For example, if the victim is logged into a social network account, the worm has the ability to modify the victim's home page and can send messages to the victim's friends.

: *Step 3 – Replication:* The worm now replicates itself. As shown in Figure 3.12(a), the JavaScript worm uses its victim's own privileges to send the social network web server a request to change the victim's home page. The victim's home page is now altered to include a copy of the Javascript worm.

: *Step 4 – Propagation:* When other benign users subsequently visit the infected victim's page, Steps 2 and 3 are repeated. Such a strategy has been demonstrated in the wild to support worm epidemics that can grow beyond a million infections.

3.2.4. Design

Here, we first provide an overview of the approach taken by PathCutter. Next we define the concept of views and describe strategies used by PathCutter for implementing view isolation and action authentication. Finally, we describe how view isolation and action authentication can prevent the propagation of an XSS worm.

3.2.4.1. Design Overview. PathCutter first isolates different pages from the server at the client side, and then authenticates the communication between different pages and the server. By doing this, the worm propagation path, in the form of an unauthorized request from a different page will be successfully blocked. The two main self-propagation routes of an XSS worm are cut off as shown in Figure 3.12(b).

- **Malicious HTTP request to the server from the infected page.** This is the most common exploit method employed by XSS worms, i.e., they send a request to modify the benign user's profile/contents at the server from the attacker's (or infected user's) page. Because the request is from the victim's client browser, the server will honor that request. In our system, because the originating page of each request will be verified, the server can easily deny such a request.

- **Malicious DOM access to the victim's page from infected page at client side.** An XSS worm can modify the victim's page at the client side to send a request on behalf of that page. Because pages are well isolated at client side, this self-propagation path is cut off.

Key Concepts. Key concepts used in PathCutter are defined as follows.

- **Views.** A view is defined as a portion of a web application. At client side, a view is in the form of a web page or part of a web page. As a simple example, one implementation at a blogging site might consider different blogs from different owners to be different views. It might also consider comment post forms to be a separate view from the rest of the page.

- **Actions.** An action is defined as an operation belonging to a view. For example, a simple action might be a request from blog X (view X) at client side to post a comment on X's blog post.

- **Access Control List (ACL) or Capability.** Access control list records all the actions that a view can perform. In the previous example, a request from X cannot post on blog Y's blog post, because X does not have the right to do this action. Capability is a secret key that a view owns that enables it to perform a specific action. Our system supports the use of either ACLs (in the form of referrer-based validation) or capabilities (in the form of per-url session tokens) for access control.

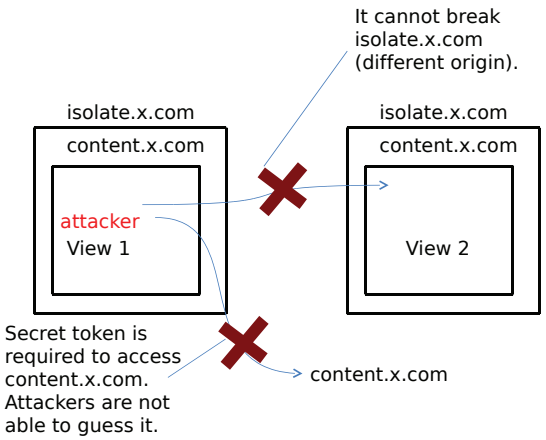

Figure 3.13. Isolating Views Using Psuedodomain Encapsulation

3.2.4.2. Web Application Modification (View Separation, Isolation and Authentication). We explore and evaluate different strategies for implementing view separation, view isolation and view authentication, and securing an application using PathCutter.

Dividing Web Applications into Views. We imagine that there are at least three potential strategies for separating web application content into views. First, a web application can be divided into views based on semantics. For example, a blog web site can be divided using blog names. Forums can be divided based on threads and subject matter. A second way to divide web applications is based on URLs. For example, when clients visit `blog.com/options` and `blog.com/update`, we can consider those two to be from different views. Finally, in some web applications, user-injected contents like comments might be on the same web page as vulnerable actions such as posting comments. In such cases, we need to isolate either those user comments or the vulnerable actions.

Isolating Views at the Client Side. According to the same-origin policy (SOP), DOM access for different sessions from the same server is allowed by default. Theoretically, we can isolate each view inside a new domain. But numerous domain names are required. PathCutter **encapsulates**

views within a pseudodomain to achieve isolation by just two domain names. As shown in Figure 3.13, for each view from contents.x.com, we embed an iframe with pseudodomain name *isolate.x.com* inside the main page. Therefore, an attacker who obtains control of *contents.x.com* in one view, cannot break *isolate.x.com* to access contents inside another view that also belongs to *contents.x.com* due to the same-origin policy. HTML5 also provides a *sandbox* feature for preventing the origin access that can be used to further strengthen isolation between different views.

Authenticating Actions. PathCutter checks the originating view for each action (e.g., posting a comment) to ensure that the view has the right to perform the specific action. Either of the following two strategies might be implemented to authenticate actions.

- **Secret Tokens.** We could explicitly embed a secret token with each action or request, especially those tending to modify contents on the server side, as a *capability*. A simple request might look like the following:

  `http://www.foo.com/submit.php?sid=****&...`

  The server will check the sid of each request to see if it has the right to modify the contents. As an attacker will not be able to guess the value of the secret token (sid), a request from the attacker's view will not have the right to modify contents on another user's page.
- **Referrer-based View Validation.** The referrer header in the HTTP request can be used for recognizing views from which an action originated. Then servers can check if the action is permitted in the *access control list*. If not, the action is denied.

3.2.4.3. Severing Worm Propagation. For a JavaScript worm that seizes control of a certain view of an application by exploiting an XSS vulnerability, there are two possible avenues to propagate as shown in Section 3.2.4.1. Blocking the worm propagation can be considered in terms of blocking

http://www.foo.com/blog1/index.php:

```
<iframe src="contents.foo.com/blog1/index.php?token=**"
 sandbox="allow-forms, allow-scripts">
</iframe>
```

Figure 3.14. Implementing Session Isolation in WordPress

the following two forms of malicious behavior. First, the worm can initiate an illegal action to the server in order to exploit other views. Because PathCutter checks every action originating from each view, illegal actions will be prevented. Second, the worm can open another view at client side, and then infect that view by modifying its contents. PathCutter's view isolation logic ensures that the worm cannot break boundaries of different views belonging to a web application at the client side.

3.2.5. Implementation

3.2.5.1. Case Study 1: Server-side Implementation - WordPress. We use WordPress [**49**], an open source blog platform, as an example to illustrate the feasibility of implementing PathCutter by modifications at the server side. We find that just 43 lines of additional code were required to add support for secret token authentication and view isolation. It took the authors less than five days to understand WordPress source code and insert those modifications.

Dividing and Isolating Views. We enable the multisite functionality of WordPress, and our implementation classifies different blogs in WordPress as belonging to different views. For example, `www.foo.com/blog1` and `www.foo.com/blog2` will be divided into different views. A finer-grained separation of views, such as different URLs, can also be adopted. As a proof of concept, separation by different blogs is implemented. As shown in Figure 3.14, a view will be isolated at client side by iframes. Every time a client browser visits another user's blog, the real

```
document.onload = function() {
  forms = document.getElementsByTagName("form");
  for (i=0; i<forms.length; i++) {
     forms[i].innerHTML="<input type=\"hidden\" value=\""
               +window.mySID+"\"/>" +forms[i].innerHTML;
  }
}
```

Figure 3.15. JavaScript Function Added to WordPress for Inserting Secret Tokens into Actions

contents will be embedded inside the outer frame to achieve isolation. Borders, paddings, and margins will be set to zero in order to avoid any visual differences.

Identifying Actions. Vulnerable actions in WordPress are normally handled by a post operation in a $form$ tag. For example, the *comments posting* functionality is the output to a user through *comment-template.php* and handled in *wp-comments-post.php*. Similarly, the *blog posting/updating* functionality is the output to a user through *edit-form-advanced.php* and handled in *post.php*.

Authenticating Actions. We use capability-based authentication (using a secret token) to validate user actions. Every action belonging to comment or blog posting categories must be accompanied by a capability, or else the action will be rejected. We implement this by injecting a hidden input into the form tag, as shown in Figure 3.15 by JavaScript, such that the client's post request to the server always includes a capability.

The ideal locations for implementing authentication are at points where client-side actions affect the server database. WordPress has a class for all such database operations and because every database operation will go through that *narrow interface*, we can quickly ensure that our checks are comprehensive.

3.2.5.2. Case Study 2: Server-side Implementation - Elgg. Elgg [**14**] is an open social network engine with many available plug-ins. We use Elgg 1.7.10 with several basic embedded plug-ins

Original code:

```
echo elgg_view('input/form', array('body' => $form_body,
'action' => "{$vars['url']}action/comments/add"))
```

After PathCutter modification:

```
echo "<iframe style = 'background:inherit;border:0;margin:0;padding:0'
        sandbox='allow-forms' scrolling='no' height='400pt' width='100%'
        src='http://other.com/echo.php?content="
     .urlencode(elgg_view('input/form', array('body' => $form_body,
     action' => "{$vars['url']}action/comments/add"))."'/>";
```

Figure 3.16. Isolating Views in Elgg by Modifications to edit.php in views/default-/comments/forms/

such as friends and blogs. Just two additional lines of code were required to add support for view isolation into the Elgg source code base. An additional file was also required to support the modification which had 23 lines. It took the authors less than three days to understand the Elgg source code and insert the corresponding modifications.

Dividing and Isolating Views. As discussed below, Elgg has built-in mechanisms to protect the *post* action. However, a JavaScript worm can still steal the secret token just as in the case of the MySpace Samy Worm. For example, the worm could send an XMLHttpRequest to the server to get the posting page and then steal the token. Therefore, we need to isolate specific views at the client side to protect the secret token as shown in Figure 3.16. Instead of using a div to submit a comment, we adopt methods mentioned in Section 3.2.4.2 to isolate the view for posting comments.

Identifying Actions. The action we wish to protect is the comment posting action in blog functionality of Elgg. It is handled in mod/blog/actions/add.php.

Authenticating Actions. The blog plug-in functionality in Elgg has already implemented action authentication. A secret token, named $\_\_elgg\_token$, is embedded in each post action that is checked by add.php upon each post request. If the token is incorrect or missing, an error message is returned to the user. We extend this logic to also check the *referer* header of each post action.

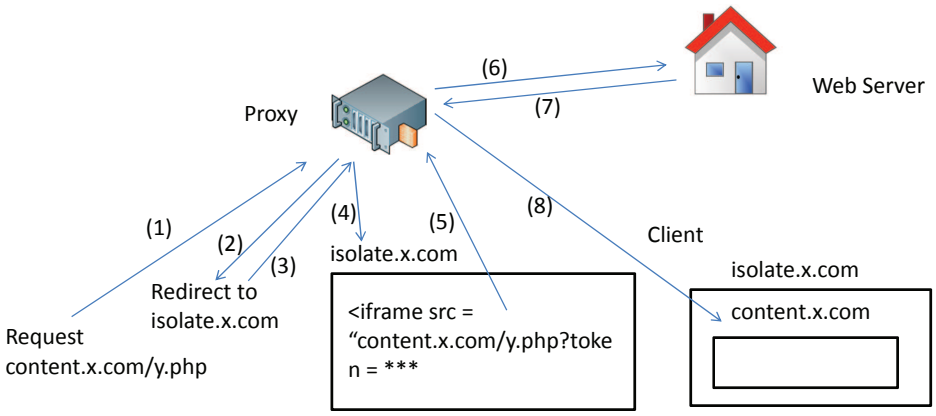

Figure 3.17. View Isolation in PathCutter Proxy Implementation

Fixing Cascaded Style Sheet (CSS) Issues. After isolating the *post* action into a separate view, we still need to fix several outstanding CSS issues, including the following. First, we make the iframe body transparent to leave the original background unaltered. Second, we include all original CSS files in order to retain the original style and layout. Finally, we make the iframe size automatic and use the seamless attribute in HTML5 to ensure that the iframe is well integrated with its parent.

3.2.5.3. Case Study 3: A Proxy Implementation. Although a server-side implementation is most desirable, a proxy-based deployment approach is attractive in certain scenarios because it provides greater flexibility. For example, this enables the service provider to deploy PathCutter without changing the application, or alternatively, PathCutter could be deployed at the client's enterprise network. Here, we describe a prototype implementation of PathCutter over Privoxy [**35**] that secures Facebook.

Isolating Existing Views. As shown in Figure 4.3, when the client browser requests a URL, such as content.x.com, which requires isolation, the proxy redirects the URL to isolate.x.com with an embedded iframe containing a secret token in the src property. When the client browser requests the redirected URL, the proxy forwards the request to the real web server and displays the returned content in the iframe.

Original code:

```
<span data-jsid="text"> user comments </span>
```

After PathCutter modification:

```
<span data-jsid="text">
<iframe scrolling='no' height='100%' sandbox style='..'
 src='http://foo.com/echo.php?content=user%20comments'/>
 </span>
```

Figure 3.18. Implementing View Separation for Facebook

Dividing Nonexisting Views. As an example, we consider an individual's Facebook page that typically includes comments and updates from other users in the individual's friend circle. Hence, we need to isolate multiple views within each page. When we look at the HTML source code of Facebook, we find that each comment and update is embedded inside a `span` tag. So, the PathCutter proxy identifies span tags and simply uses a regular expression to replace them with an iframe. For example, as shown in Figure 3.18, we replace the `span` tag with an iframe and echo back the other users' comments. Even if a malicious script is injected by an attacker, our transformation ensures that the malicious script operates in a separate view, isolated from the view with the capability to modify the victim user's content.

Our proposed approach to dividing views at the proxy is vulnerable to *injection attacks*, i.e., the attacker can inject the same pattern that we are looking for into the comments. For example, as shown in Figure 3.18, the attacker can inject `<span data-jsid="text">` or `</span>` to confuse the proxy. The proxy needs to *modify the signature* in order to deal with such injection attacks. For example, if the attacker tries to inject `</span>`, the proxy needs to find the last matching `</span>` instead of the first match.

Authenticating Actions. The proxy checks the *referer* header of each request. If the request is from *foo.com* (our echoing server), this indicates that it is potentially a forged request originating from a malicious comment. Hence, such requests are rejected.

Allowing DOM access from Flash:

```
XN.template.flash=function(o){
return  <embed src=\"+o.filename+\ type=\application/x-shockwave-flash\
+width=\"+(o.width||320)+\ height=\"
+(o.height||240)+\ allowFullScreen=\true\ wmode=\"
+(o.wmode||transparent)+\ allowScriptAccess=\always\></embed>;
};
```

Modifying DOM to add and invoke the malicious script:

```
var fun = var x=document.createElement(SCRIPT);
x.src=http://n.99081.com/xnxss1/evil.js;
x.defer=true;document.getElementsByTagName(HEAD)[0].appendChild(x);;
flash.external.ExternalInterface.call(eval,fun);
```

Figure 3.19. Flash Vulnerability Exploitation in the Renren Worm

3.2.6. Evaluation

We analyze the effectiveness of the PathCutter approach against five real-world worms. We further evaluate the server-side implementation against two proof-of-concept worms.

3.2.6.1. Evaluation against Real-world Worms. We evaluate PathCutter against two server-side XSS (MySpace Samy, Yamanner) worms and three client-side XSS (Renren, SpaceFlash, Boonana) worms by analyzing the source code and online descriptions of these worms.

1. Boonana Worm. Boonana [**54**] is a Java applet worm that was released in October 2010. The propagation of this worm can be divided into the following steps:

(1) *A benign user visits an infected profile with a malicious Java applet posted by the attacker.*

(2) *The malicious Java applet exploits a Java vulnerability to inject malicious JavaScripts on the client side, thus escalating its privilege to the victim user.*

(3) *The worm posts itself on the visiting user's wall using the stolen cookie.*

(4) *Boonana proliferates over the social network when more people visit the malicious applet.*

PathCutter Defense: PathCutter blocks the Boonana worm propagation at Step 2, by ensuring that the worm gains only the privilege of a view containing the page of the malicious Java applet,

and not the privilege of the user's Facebook profile page. Therefore, the web server declines the request to post on the user's wall.

2. Renren Worm. The Renren worm[**53**] was a Flash-based worm that spread on the Renren social network (one of the largest Chinese social networks). The worm was released in 2009 and affected millions of users. The propagation of this worm can be divided into the following steps:

(1) *A victim user visits an infected profile with a malicious Flash movie posted by the attacker.*

(2) *The malicious Flash movie exploits a Flash vulnerability, which injects malicious JavaScripts at the client side, thus escalating its privilege to that of the victim* (as shown in Figure 3.19).

(3) *The injected script replicates itself on the victim's wall.*

(4) *When other users visit the infected user's profile, the worm repeats the infection and replication process, and thus spreads.*

PathCutter Defense: A user who wants to share something on the Renren social network, needs to get a page `http://share.renren.com/share/buttonshare.do?link=..`, and then send the real share request. PathCutter isolates the real sharing request in a view A that is different from view B where updates from friends are displayed. Therefore, at Step 2, the worm obtains only the privilege of that specific view B, and so is unable to replicate itself on behalf of the victim user.

3. MySpace Samy Worm. The MySpace Samy worm [**31**] was one of the first cross-site scripting worms that spread in the MySpace social network, affecting over a million users in 2005. The attack steps of Samy worm are as follows:

(1) *The victim visits an infected profile page, which carries a malicious script (due to a script filtering problem in MySpace). The infected user's profile has the following code to embed a malicious* $< div\ style = background : url('java\backslash nscript : eval(...)')$

(2) *The worm first steals the secret token, required by MySpace, using a GET HTTP request to escalate its privilege to the view that can send a POST HTTP request.*

(3) *The worm posts itself to* `/index.cfm?fuseaction=profile.previewInterests&token=**` *on the victim's profile via XMLHttpRequest.*

(4) *The Samy worm proliferates over the social network as more victims visit the growing list of infected profiles.*

PathCutter Defense: Two propagation paths are severed. First, when the worm tries to steal the secret token required by MySpace, that access is denied because different profiles are isolated into different views. The XMLHttpRequest is sent to a different domain and the response is not accessible by the worm. Second, when the worm sends out the POST request, that request is actually from the infected user's profile and not from the victim's profile. PathCutter correctly checks the capabilities of the originating view and denies such modifications.

4. SpaceFlash Worm. The SpaceFlash worm [**42**] was released in 2006 as another JavaScript worm spreading on the MySpace network by exploiting a Flash vulnerability. The steps of a SpaceFlash infection are as follows:

(1) *A victim user visits the attacker's "About Me" page with a malicious Flash applet.*

(2) *The malicious Flash applet is executed, exploits a Flash vulnerability to access the MySpace page, and retrieves the victim's profile by visiting* `http://editprofile.myspace.com/index.cfm?fuseaction=user.HomeComments`*, thus escalating its privilege to match the victim.*

(3) *The worm sends out an AJAX request to the server to post itself on the victim's "About Me" page.*

(4) *SpaceFlash proliferates over the social network as more victims visit the growing list of infected "About Me" pages.*

<form<?php echo $enctype; ?> id="upload-file" method="post" action="<?php echo get_option('siteurl')
. "/wp-admin/upload.php?style=$style&tab=upload&post_id=**$post_id**"; ?>">

Figure 3.20. CVE-2007-4139 (Untainted Input in wp-admin/upload.php)

PathCutter Defense: In step 2, the worm cannot escalate its privilege and therefore the unauthorized post request in Step 3 is rejected, as it is does not originate from the victim's "About Me" page.

5. Yamanner Worm. The Yamanner worm [**26**] was released in 2006 and infected tens of millions of users. It was a JavaScript worm spreading in Yahoo! mail. The steps of infection and propagation were as follows:

(1) *A victim user receives malicious email from the attacker.*

(2) *The victim user clicks on the email and the malicious scripts inside the email are executed due to a bug in Yahoo's script filter. Using these scripts the worm acquires the victim's privilege.*

(3) *The worm opens the victim's address book and sends out malicious email containing itself to those who are listed in the book.*

(4) *Yamanner proliferates across the email social networks of those victims who open the email.*

PathCutter Defense: Even though the worm logic gets executed at the client, it does not have the privilege of sending email to others. In the PathCutter approach, a secret token is required to perform the action and the worm cannot steal the token because it is isolated inside a different domain.

Summary. Although the five worms described above all use different vulnerabilities and techniques to achieve JavaScript execution privileges on behalf of a victim at a specific social network web site, they are all blocked by PathCutter. PathCutter exploits the fact that aforementioned worms all have to send a request to the server from an untrusted view in order to post themselves on the victim's profile. The common propagation path is severed in each case.

```
check_infected();
// check if the user is infected or not
xmlhttp = new XMLHttpRequest;
xmlhttp.open("POST", post_url,true);
xmlhttp.onreadystatechange=function() {
  if (xmlhttp.readyState==4) {
    set_infected();
  }
}
xmlhttp.setRequestHeader("Content-type"
    , "application/x-www-form-urlencoded");
xmlhttp.setRequestHeader("Content-length"
    , payload.length);
xmlhttp.send(payload);
```

Figure 3.21. Propagation Logic of Custom Proof of Concept Worm

3.2.6.2. Evaluation against Experimental Worms. We evaluate our implementation using experimental worms that operate on WordPress and Elgg. We implemented two XSS attacks based on a published vulnerability in WordPress (CVE-2007-4139 stored XSS [**11**]) and a custom DOM-based attack on Elgg that is based on a proof-of-concept DOM-based attack [**12**]. We adopt the same code as published in [**12**] to let Elgg read language settings in URL parameters and write it on the web page without sanitizing. Figure 3.20 shows the WordPress XSS vulnerability, which does not properly cleanse one input from the user.

To convert the proof-of-concept attacks into worms, we incorporated two propagation modules: a custom propagation module that we developed and a published worm template [**29**]. The custom propagation module implements a simple worm to propagate on the network as shown in Figure 3.21. The functionality of the worm is to post itself (for DOM-based XSS attack, it will be the URL with code injected into language settings) on the victim user's blog comments by AJAX when the victim visits an infected page. The published worm template is the universal JavaScript XSS worm template [**29**] as shown in Figure 3.22. We updated the worm template with the XSS vulnerability we created, and the worm replicates itself as blog comments.

```
  //Worm's HTTP request object
function xhr() { ... }
  Object.prototype.post = function(uri,arg) {
   /*** usage: xhr().post('foo.php'); ***/
   this.open('POST', uri, true);
   this.setRequestHeader('Content-type'
    ,'application/x-www-form-urlencoded');
   ...
   this.send(arg);
  };
  /*** source morphing component ***/
  Object.prototype.morph = function(s) {
   ...
   switch(morphtype) {
     case "unicode": ...
     case "charcodes": ...
   }
 }
```

Figure 3.22. Propagation Logic of a Published Worm Template

For evaluation, we deployed WordPress and Elgg with and without our modifications on a Linux machine with Apache-PHP-mySQL installed in our network. Before integrating PathCutter, we found that both worms propagate easily in both networks by replicating themselves in the form of blog comments. After adopting PathCutter, worm propagation is effectively stifled, as view separation ensures that comments must be posted from the same view as that of the victim's blog.

3.2.6.3. Performance Evaluation. We summarize memory and rendering time overhead introduced by PathCutter. No measurable CPU overhead was introduced by the system at the client.

Memory Overhead. The memory overhead introduced by PathCutter depends on the complexity of pages on the social network and strategy used by PathCutter to separate views. In the Elgg and WordPress examples of Section 4.1.4, we chose to isolate html elements with *vulnerable actions* into separate views. Hence, only two frames were required per blog page, and the memory overhead we introduced was negligible.

Instead, if we choose to isolate views based on *content provenance*, like comments in the Facebook example in Section 4.1.4, the memory overhead we introduce depends on the number

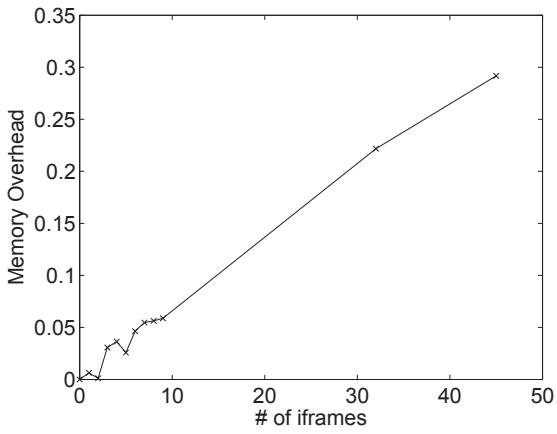

Figure 3.23. Memory Overhead for Isolating Comments from Friends

of comments on the web page. We conducted an experiment where we visited Facebook, using our proxy implementation, on a Linux client running Firefox 3.6.18 with 2 GHz dual core Xeon processors and 2 GB of memory. The results are shown in Figure 3.23. We find that when the number of comments is less than 10, PathCutter's view isolation iframes introduce less than 10% overhead. If we have 45 comments, the iframes introduce nearly 30% overhead. On Facebook threads are folded by default when the number of comments in a thread exceeds three. Therefore, we do not expect the overhead to be significant. Finally, if Facebook adopts PathCutter's approach and isolates potentially dangerous actions (e.g., comment posts) into a different view, we do not need to isolate comments at the proxy and introduce those overheads.

Rendering Time Overhead. We perform a rendering time comparison between our modified Elgg and the original Elgg implementation. The experiment is performed on a Linux machine running Firefox 3.6.18 with 2 GHz dual core Xeon and 2 GB memory. We use Firebug to monitor the onload event. Experiments are performed ten times for the blog posting page of each version. The average rendering time for modified Elgg is 1.18 seconds, and for original Elgg is 1.14 seconds. The additional rendering time overhead is about 3.5%.

CHAPTER 4

Connecting Different Principals

4.1. A Communication Channel to Facilitate Web Single Sign-on

The proliferation of web applications on the Internet has led to a sharp increase in the number of accounts (and corresponding credentials) that a modern web citizen has to create and use, and whose authentication credentials the user has to remember. Inconveniences stemming from having to keep track of these accounts, and the tendency of web sites to suffer from security breaches that would expose user credentials resulted in a push to establish *single sign-on* (SSO) systems. As the name implies, these systems help to reduce the large numbers of account credentials, and replace these credentials with a single central identity with which a user can authenticate to many different web sites.

SSO systems allow a user to sign into a web site (*Relying Party* or RP), such as StackOverflow, with authentication provided by an *Identity Provider* (IdP), such as Facebook or Google. While this greatly increases the convenience for both the users and the web site operators, it also provides new opportunities for attackers. A recent analysis of web SSO by Wang et al. [**155**] identified five vulnerabilities in which a malicious attacker can impersonate a benign RP or intercept the communication between a benign RP and the IdP, leading to a compromise of the user's security. In fact, further research by Sun et al. [**144**] has shown that 6.5% of RPs[1] are actually vulnerable to such impersonation attacks.

[1]The original text states "we also found that 13% of RPs use a proxy service from Gigya, and half of them are vulnerable to an impersonation attack."

To further understand SSO vulnerabilities, formal security analysis has been carried out on existing SSO protocols. For example, AuthScan by Bai et al. [**58**] extracts the protocol specification from a provided SSO implementation and verifies the specification using a formal analysis tool. Another approach, ExplicatingSDK [**156**] leverages formal analysis combined with a semantic model to identify hidden assumptions in the designs of the software development kits (SDKs) underlying many SSO implementations. Similarly, InteGuard by Xing et al. [**158**] correlates program invariants to find and mitigate logic flaws in SSO specifications by using a proxy situated between the users and the IdP. However, none of these prior approaches identified the actual root cause of SSO vulnerabilities, *i.e.*, why these flaws exist in the current protocols in the first place. In our work, we examined the design of existing SSO protocols, and we determined that the root cause of the aforementioned vulnerabilities lies in the design of the communication channel between the RP and the IdP. Depending on the protocol implementation, this channel either suffers from being a one-way communication protocol, or from a lack of authentication. This untrustworthy channel, in turn, makes existing SSO protocols prone to RP impersonation attacks. Therefore, we propose to use a secure, authenticated, bi-directional channel between the RP and the IdP to prevent these attacks.

It is important to realize that the attacks discussed in this paper are not just theoretical. In fact, the vulnerabilities in these protocols are currently being used to subvert SSO authentication for the attackers' own purposes. These real-world, high-profile attacks have been carried out with extremely high-impact on popular web sites. Last year's attack on the SSO communication between the web site of the New York Times (the 38th most popular web site in the US), which is an RP, and Facebook, which is an IdP, is such an example [**155**]. This attack, made possible by weaknesses in the current design of web-based SSO protocols, allowed attackers to access victim accounts with the same privileges that were given to the NYTimes itself, including access to private user data

and, in some cases, the ability to post messages on the user's behalf. Similar attacks have been also carried out against the communication between Facebook and Zoho.com, the JanRain wrapping layer over Google's SSO implementation, and other web sites that rely on SSO protocols.

Thus, the solutions presented in this paper are not just *theoretically interesting*, but are actually *practically relevant* to solving an extremely pressing problem in web security. We designed both a practical replacement for vulnerable protocols, which currently undermine modern web SSO. Additionally, to enable a smooth transition and provide time for our design to be adopted by the industry, we introduce a *proxy* that can be deployed by web sites (RPs, in the current model) to secure existing SSO protocols. It is our hope that this proxy will ease the adoption of our secure SSO design and eliminate current vulnerabilities.

In this paper, we make the following technical contributions:

- *Dedicated, Bi-directional, Authenticated, Secure Channel.* Utilizing an existing in-browser communication channel, we establish a dedicated, bi-directional, authenticated, secure channel between RPs and IdPs. By leveraging public-key cryptography, an RP and an IdP authenticate each other and share a common secret, *i.e.*, the session key, between each other. All further communication between the RP and the IdP is then encrypted with this session key, and is kept secret from any potential eavesdroppers.
- *Flexible SSO Protocol over the Secure Channel.* The aforementioned secure channel provides flexibility for SSO protocol designers. Specifically, our channel design provides a secure platform for protocol architects to further customize. In contrast, existing, verified protocols, such as the Secure Electronic Transaction (SET) [**38**] protocol, require customers and users to strictly follow the protocol specification. To show the flexibility of our approach, we discuss example implementations of the most popular SSO protocols: an OAuth-like and an OpenID-like protocol that utilize our channel.

- *Formal Verification of the Channel.* Ultimately, we formally verify our channel design with the verifier ProVerif by Blanchet [**36**]. We further validate this verification step by manually inserting a vulnerability in the channel design and showing that ProVerif discovers this vulnerability.
- *Performance Evaluation.* We implemented and evaluate a prototype system. In our evaluation (c.f.,Section 4.1.5), we show that the overhead our approach to SSO introduces about 650ms latency (given a network delay of 50ms to reflect the latency observed at standard, residential Internet connections), which is generally acceptable for common users. In addition, we also provide a detailed breakdown of the latency of each interaction of our SSO implementation.
- *Gradual Deployment.* Apart from our clean-slate design, which would be deployed by the IdP, we also introduce a proxy that acts as a "fourth-party" IdP. This proxy accommodates and protects existing SSO protocols and, by this, allows for an easy transition to our our more secure SSO protocol. Aiming for broad adoption and a general solution, we designed our proxy so that it can be deployed by a legacy RP or a legacy IdP. From the viewpoint of a legacy IdP, our proxy acts as a RP retrieving users' information in a controlled, secure environment; from the viewpoint of the legacy RP, it acts an IdP relaying users' information it retrieved from the real IdP. To guarantee the various security properties that must hold to ensure protection against impersonation attacks with respect to the threat model, we formally verify the design of our proxy.

4.1.1. Threat Model

In this section, we introduce several key concepts relating to web SSO and present several known attacks against current protocols and implementations. For completeness, and to provide a better understanding of the threat model, we discuss attacks that are in the scope of this paper, as well as attacks that we deem as out of scope.

4.1.1.1. Concepts. Generally, three entities are involved in an instance of an SSO protocol. The three participating parties are:

- *Identity Provider (IdP).* An identity provider provides a centralized identification service for its users. Examples of identity parties are OpenID [**32**], Facebook Connect [**92**], and Google's single sign-on implementation. Additionally, there exist several aggregation services, such as JanRain Engage [**21**], which handle single-sign-on services for multiple IdPs. In the latter case, both JanRain Engage and the original IdP act as IdPs.
- *Relying Party (RP).* A relying party is a web site that makes use of the services an IdP provides to authenticate its users. The way this is accomplished differs from protocol to protocol. For instance, for an OpenID-like service, an RP acquires the user's identity with an IdP's signature; alternatively, for an OAuth-like service, an RP acquires a token or key from the user (who interacts with the IdP), which can then be used to fetch additional information from the IdP.
- *User.* A user is a client of both the IdP and the RP. The user maintains a single sign-on identification on the IdP's web site. In our threat model, users are benign; in other words, we do not consider cases initiated from an attacker's browser, and we require that the same-origin policy is enforced in the user's web browser.

4.1.1.2. In-scope Attacks. Recent research by Wang et al. [**155**] identified many *identity impersonation attacks* of SSO protocols, attacks where an adversary manages to fake the identity of parties involved in the SSO protocol. In our threat model, we only consider the case where an attacker is able to impersonate an RP in an attempt to steal user information. In fact, these attacks account for five of the eight confirmed attacks on SSO protocols identified by Wang et al. The remaining three cases, we deem as out-of-scope, and we discuss the reason for this decision in more detail in Section 4.1.1.3. We further classify the five in-scope attacks into two categories:

- *A malicious RP initiates the attack.* For instance, in a vulnerability found between the New York Times (NYTimes) and Facebook [**155**], a malicious site could pretend to be the NYTimes and initiate the SSO process, *i.e.*, an attacker simply sends a request using the application ID of the NYTimes (`app_id` in Listing 4.1) to pretend to be the NYTimes.
- *A benign RP initiates the request, but a malicious RP receives the response.* For instance, in a vulnerability found in the interaction between Zoho.com and Facebook [**155**], Zoho.com initiates the communication to Facebook, but a malicious party receives the response from Facebook, *i.e.*, an attacker changes the `redirection_url` or the `next_url` in Listing 4.1 to receive the response that contains the access token.

```
GET https://www.idp.com/login?app_id=****
    &redirection_url=https://www.idp.com/granter?
    next_url=https://www.rp.com/login
Host: www.idp.com
Referer: https://www.rp.com/login
Cookie: ****
```

Listing 4.1. Example of a HTTP Request from an RP to an IdP. Here, a malicious RP could initiate the communication by using a benign RP's app_id or intercept the communication by changing the redirection_url or next_url parameter.

4.1.1.3. Out-of-scope Attacks. Since SSO is a broad topic, there are some categories of attacks that are out-of-scope for this paper. Here, we provide an example list of such attacks that we deem out-of-scope, and why we decided to not take them into account. Generally, many of the attacks listed can already be mitigated leveraging prior work and we list them simply for completeness.

- *Phishing.* We do not consider social-engineering attacks, such as the phishing of a user's credentials, because, in our opinion, the prevention of phishing is more of an educational or user-interface issue than a protocol issue.
- *Compromised or vulnerable RP.* Bhargavan et al. [**69**] consider that an RP might be compromised and propose defensive JavaScript techniques to prevent untrusted browser origins. On the contrary, in our threat model, a benign RP remains integral and uncompromised, but the communication channel between the benign RP and the IdP is vulnerable so that a malicious RP can control the channel.
- *Malicious browser.* Some of the discovered attacks occur inside of malicious user's browser. For example, in an example involving Google ID [**155**], an attacker is able to log into a user's account on the attacker's machine because the RP does not check whether the email field is signed by the IdP.
- *Implementation issues.* Some of the discovered attacks are because of different interpretations of the protocol that is used between an RP and an IdP. For example, in a vulnerability involving PayPal [**155**], the RP (PayPal) and IdP (Google) treat a data type differently.
- *Privacy leaks.* Uruena et al. [**150**] identify possible privacy leaks to third party providers (such as advertisers and corresponding industries) in the OpenID protocol.

Generally, we consider all these attacks as out of scope because they are due to a user error, an implementation error (specific to improperly-implemented RPs or IdPs), or do not involve any communication between an RP and an IdP.

4.1.2. Revisiting Existing SSO Designs and Attacks

In order to identify the root cause of the identity impersonation attacks presented by Wang et al. [**155**], it is critical to understand some aspects of existing SSO designs, especially the communication protocol between the RP and the IdP. Wang et al. [**155**] abstracted this communication by introducing the concept of a *browser-relayed message* (BRM). A BRM describes a request from either an IdP or an RP, to either the IdP or RP respectively, together with the resulting response. While this abstraction is already quite useful in detecting vulnerabilities, we feel that it has no bearing on the root cause of the vulnerabilities that exists in existing SSO designs. Instead, we abstract the protocol differently: the RP simply communicates with the IdP through an established channel. In this context, two questions remain: (i) what is the identity of the parties involved (authenticity), and (ii) how do these parties communicate to achieve confidentiality and integrity?

4.1.2.1. Identity. Generally, three parties are involved in an attack. Understanding how each party is identified to the other parties is a prerequisite to analyzing the root cause of impersonation attacks.

- *IdP.* The IdP is generally identified by its web origin, *i.e.*, <scheme, host, port>.
- *User.* The user is often identified by a unique identifier, such as their username or email address. While the identification of users can cause some confusion in an SSO protocol, attacks resulting from this confusion are out of scope because they are generally implementation or documentation errors. In the rest of this paper, we assume that there exists a correct and unique identifier for each user.
- *RP.* The identity of an RP can vary according to protocols imposed by different IdPs. For example, Facebook Connect uses the identifier "app_id" to identify an RP, while JanRain chose to adopt "AppName" and "settingsHandle" as the RP's identifier [**155**].

Recent attacks also show us that the identity of a benign RP can be easily forgeable by a malicious RP. For instance, in the case of Facebook and the NYTimes [**155**], the malicious RP forged the identifier of the NYTimes. Because of this, we require an unforgeable identifier to represent the identity of an RP. In the same spirit as for the IdP, one can use the web origin to be tracked in the client browser. Since web origin tracking is already a basic security property that is enforced by all modern browsers, it is very hard for a malicious RP to forge it.

4.1.2.2. Communication between the RP and the IdP. Simply equipping the RP with an unforgeable identifier, however, does not mitigate existing vulnerabilities. In addition, during the execution of an SSO protocol, one must verify the identity of the RP at every step. To detail the problems caused by this, we carefully re-examine the communication between the RP and the IdP via a client's browser in existing protocols. We classify those interactions into two main categories: HTTP(s) request to a third-party server and an in-browser communication channel.

HTTP(s) Requests to a Third-Party Server. In the first category, comprising OpenID, Security Assertion Markup Language (SAML) [**39**], and AAuth [**147**], the RP and the IdP communicate via HTTP(s) requests with each other. The detailed process is as follows: consider an RP trying to connect to an IdP. First, the RP's JavaScript code running in the client's browser sends a request to the RP. The RP sends a response containing an HTTP 3xx redirection (or, alternatively, some kind of other redirection, for instance, via automatic form submission) to the client. Third, based on this redirection, the RP's code in the client browser sends a request to the IdP. Finally, the browser communicates with the IdP and completes the authentication process.

Problem: This interaction via third-party HTTP(s) requests is a one-way channel, *i.e.*, after an RP talks to an IdP, the IdP cannot send a response back. In order to actually receive a response, the RP needs to tell the IdP where to forward the client's browser upon the authentication's completion. Generally, this is done by utilizing a parameter in the request, such as the `next_url` parameter

of Listing 4.1. The main issue here is that this parameter could be modified by an attacker, and, in turn, can result in an identity impersonation attack. To mitigate this vulnerability, a bi-directional communication channel is necessary.

In-browser Communication Channel. The second category describes protocols that communicate via an in-browser communication channel. This includes Facebook Connect[2] [**92**] and other protocols in which the RP and the IdP use JavaScript in the client's browser to communicate with each other. The different parties communicate with each other using one of a number of mechanisms, such as postMessage [**66**], URI fragments [**66**], or even Flash objects [**155**]. A detailed overview of an example protocol is shown below. In this example, the protocol makes use of the postMessage functionality, where an inline frame R from the RP communicates with an inline frame I from the IdP. To communicate, R uses `I.postMessage(Msg, IdP's origin)`, and, in turn, I receives an onMessage event. After I receives the message, it can use then use `event.source.postMessage(Msg, verified RP's origin)` to respond.

Problem: There are two issues with this approach. First, the in-browser communication channel is an undedicated, bi-directional channel without proper authentication. For each message exchanged between the RP and the IdP, the origin needs to be verified independently. Recently, researchers established that this verification step is frequently forgotten by developers [**140, 98**], and this omission can make the protocol vulnerable to impersonation attacks. For instance, Hanna et al. [**98**] determined that two prominent SSO protocols, Facebook Connect and Google Friend Connect, exhibit this problem. More recently, Son et al. [**140**] identified 85 popular web sites that were using the postMessage API incorrectly. This demonstrates the requirement for a dedicated channel with authentication for RP-IdP communication.

[2]Facebook Connect mixes the usage of HTTPs requests to third-party server and an in-browser communication channel.

Second, the in-browser communication channel is insecure. In the vulnerability between Facebook and NYTimes [**155**], attackers exploited this fact to eavesdrop on the in-browser communication between the IdP and the RP and intercept users' access tokens as the authentication takes place. Once the attacker obtains this access token, he can successfully impersonate the NYTimes to Facebook. In that case, the channel insecurity was introduced by the use of Flash objects. However, even if the communication channel is changed from Flash objects to postMessage, Barth et al. [**66**] and Yang et al. [**160**] have demonstrated that the postMessage communication could still be insecure. Additionally, Cao et al. [**79**] show that the postMessage vulnerability originally proposed by Barth et al. still exists in modern browsers for requests from the same domain, such as different blogs or applications hosted under the one domain.

To mitigate these threats, a *dedicated, bi-directional, secure channel with authentication* is required. In the next section, we discuss the steps involved in establishing such a channel.

4.1.3. Design

In the previous section, we argued for the necessity of a new, dedicated, bi-directional, secure channel between the IdP and the RP, along with a new approach to identity verification for the RP. In this section, we discuss the design of our SSO protocol, which is not subject to these shortcomings. We introduce two solutions: a clean-slate redesign and a proxy. The clean-slate design, presented in Section 4.1.3.1, must be deployed by the IdP as a new SSO service; the proxy design, described in Section 4.1.3.2, serves as a "fourth-party" IdP and is designed to be deployed by an RP that wants to protect its users from RP impersonation attacks in legacy SSO services, while still supporting authentication against these services.

4.1.3.1. IdP Deployment: Clean-slate Design. As discussed in Section 4.1.2, the core requirements of our clean-slate design of SSO are (i) the use of the web origin as its identity, and (ii)

```
parameters= {
  mWindow: parent,// optional when listening
  mOrigin: IdP/RP Origin,
  onMessage: function(msg) {//callback for msg receiving}
  onConnect:function () {//callback for creating a channel}
  onDestroy:function (){//callback for destroying a channel}
  }
//Listening to a channel
var socket = new SecureSocket();
socket.listen(parameters)
//Creating a channel
var iframe = document.getElementById("myid");
var socket = new SecureSocket();
socket.connect(parameters)
```

Listing 4.2. JavaScript Primitive for the Channel.

the use of a dedicated, secure, bi-directional channel with authentication for all communication between the RP and the IdP.

Identity Design. In our protocol, the identity of an RP is a web origin. There are two possible cases.

- *Web Origin is defined by the RP.* If an RP has its own web origin, such as the NYTimes, the RP can simply define a sub-domain for its SSO communication. For example, the NYTimes could make use of *facebookconnect.nytimes.com* for all communication with Facebook Connect to authenticate users through SSO.
- *Web Origin is defined by the IdP.* If an RP does not have a web origin, such as in the case of an application designed by a third-party web developer, the RP can adopt a sub-domain origin of the IdP (for example, *application1.connect.facebook.com*) as its identity for communication with Facebook for the authentication of users through SSO.

In either case, the RP can leverage the sandbox tag defined in HTML5 to ensure isolation of the web origin.

Communication Channel. Next, we will detail the life-cycle of a bi-directional, authenticated, secure communication channel over the following three steps: the establishing of the channel, use of the channel, and destruction of the channel.

Establishing the Channel: Handshake Protocol. To establish a secure communication channel between the RP and the IdP, JavaScript first creates a secure socket that is listening to connections from a given web origin (for example, the web origin belonging to the RP), optionally specifying a target window (such as *parent*), as shown in Listing 4.2. When the RP's web page is loaded, its JavaScript connects to the socket created by the IdP by specifying the target window (*e.g.*, a reference to an iframe) and the target origin (*i.e.*, the IdP's web origin) as shown in Listing 4.2 (Line 11-14).

The handshake protocol establishes the secure channel between the RP and the IdP, comprising an exchange their keys. The process includes the following five steps:

(1) The RP verifies the identity of the IdP and sends its public key (PK_RP) to the IdP.

(2) Upon receipt of this message from the RP, the IdP verifies the identity (web origin) of the RP.

(3) The IdP generates a session key (SK), encrypts it with the public key from the RP, and sends the encrypted session key and an encrypted partial channel number (PK_RP(N_IdP)) to the RP.

(4) The RP decrypts this using its own private key and responds with an encrypted partial channel number (SK(N_RP)). This channel number, and the channel number generated by the IdP in (3) is then stored by the browser as an index to look up the session key.

(5) With both parts of the channel number, the RP and the IdP can communicate with each other. Both N_IdP and N_RP are needed to send a message, and each message consists of a ControlByte (later used to determine the status of the channel) and the message content, encrypted with the session key.

In the aforementioned steps, N_IdP and N_RP are used to look up the session key at both the RP and the IdP, and the ControlByte is used to determine the status of the current channel, such as whether it has been destroyed. During the negotiation period, both N_IdP and N_RP are encrypted

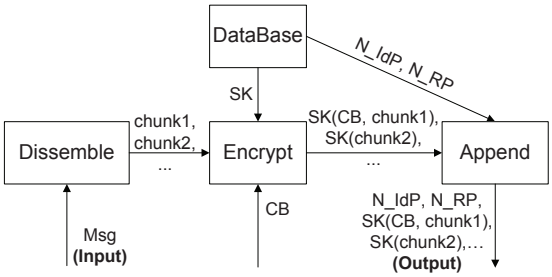

Figure 4.1. Sending a message to the channel between the client-side RP and the client-side IdP.

and protected. Later on, N_IdP and N_RP are not encrypted, and thus easily modifiable. However, if an attacker modifies N_IdP or N_RP, a different session key and message handler will be used to process the message, and will result in the message being delivered to an incorrect channel. In the absence of the correct keys, an entity listening on this channel will be unable to decrypt the message.

Use of the Channel: Sending Messages. After the channel is created, messages can be sent. When either side wants to send a message, it calls the JavaScript primitive *socket.sendMessage(msg)*, which divides the input message into small chunks (to fit the key size), appends the control byte to indicate the status of current channel, encrypts all the divided chunks with the shared session key, and appends N_IdP+N_RP to the result. This message is then sent via the in-browser communication channel, which ensures the delivery of the message. More details on this interaction are shown in Figure 4.1.

Use of the Channel: Receiving Messages. When a message is sent by one of the parties, the system first uses the N_IdP and N_RP to retrieve the session key and socket from the browser. The session key is used to decrypt the encrypted message chunks and ControlByte. These chunks are then reconstructed and delivered to the processing function on the other end of the corresponding socket. More details of this interaction are provided in Figure 4.2.

Destroying a Channel: Releasing Resources. In the process of destroying a channel, *socket.close()* is called. When either the IdP or RP calls the close method, the other side is notified.

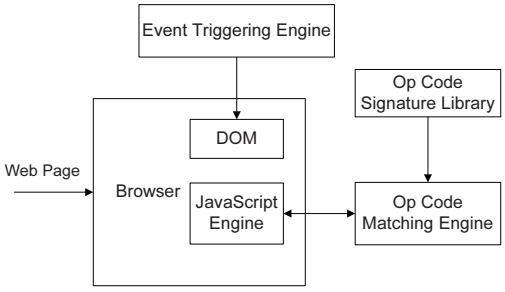

Figure 4.2. Receiving a message from the channel between the client-side RP and the client-side IdP.

For example, if an RP wants to close the channel during the communication, it sends a message with a ControlByte of 0. When the IdP receives this message, it sends an equivalent response (with the ControlByte of 0) and releases resources such as the channel number, session key, and socket. After the RP receives the response message, it also releases its resources. This is analogous to the closing of a TCP network socket.

SSO Protocol over the Channel. Our secure communication channel between the RP and IdP can now be used in the design of a secure SSO protocol. Traditionally, SSO is classified into two categories: OAuth-like protocols for authorization and OpenID-like protocols for authentication. In the former, the RP asks the IdP for an access token to fetch a user's information; in the latter, the RP asks the IdP to verify the identity of a user. We describe both paradigms in the context of our secure channel.

OAuth-like Protocol. In an OAuth-like protocol, the authorization happens through the following steps:

(1) Client-side JavaScript code served by the RP initiates a connection with the JavaScript code served by the IdP by establishing the secure channel.

(2) The RP JavaScript requests a token, which can be used to access a user's data, from the IdP JavaScript.

(3) The IdP authenticates the user, usually by having them log into the IdP's service.

(4) When this authentication is successfully completed, the IdP acquires the user's permission to allow the RP to access their information.

(5) The IdP then sends the token to the IdP JavaScript, which forwards the token to the RP JavaScript.

(6) The RP JavaScript sends the token to the RP server, which then uses that token to request the user's information from the IdP service.

OpenID-like Protocol. In an OpenID-like protocol, the authorization requires the following steps:

(1) JavaScript served by the RP establishes a secure channel to JavaScript served by the IdP.

(2) The RP JavaScript asks the IdP JavaScript to authenticate the user.

(3) The IdP server authenticates the user, usually by having them log into the IdP service.

(4) Upon successful authentication, the IdP server sends an authentication proof (typically a token encrypted with the IdP's private key) to the IdP JavaScript.

(5) The IdP JavaScript sends the authentication proof to the RP JavaScript.

(6) The RP JavaScript sends the authentication proof to the RP server.

4.1.3.2. RP Deployment - Proxy Design. We have presented a clean-slate design for SSO in Section 4.1.3.1. However, many existing, unprotected SSO implementations are currently deployed, and migrating them to our secure design will take time. To assist in securing these legacy implementations, we designed a wrapper that integrates them into our design.

As shown in Figure 4.3, this takes the form of a *proxy* (not a network proxy in its traditional meaning, but rather a "fourth-party", secure IdP), mediating the communication between an RP and IdP. The proxy acts as a legacy RP to the legacy IdP, authenticating against it like any

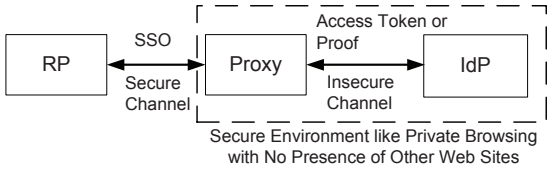

Figure 4.3. Overview of the proxy design. The secure RP is talking to our proxy through the secure channel, and our proxy is talking to the legacy IdP using a legacy SSO protocol, but contained in a secure, controlled environment.

other currently deployed RP, but with sufficient isolation to protect it from existing SSO attacks. Additionally, this proxy acts as a secure IdP to the new, secure RP.

The proxy communicates with the legacy IdP and the secure RP as follows.

Communication with the legacy IdP. In order to communicate with the legacy IdP, the proxy needs to act as an RP. For the IdPs of most existing SSO implementations (for example, Facebook Connect), we must register an application for each user, and keep the application ID secret from attackers to avoid any impersonation attacks. Specifically, only the legacy IdP and our proxy must know the application ID.

After registering an application with the IdP, the proxy performs a normal SSO authentication with the IdP and acquires the access token (in the case of the OAuth-like protocol) or proof of authentication (in the case of an OpenID-like protocol). Since these tokens incorporate the *privileges* that an RP has over the authenticating user's account, and since some subset of these privileges will be ultimately used by the secure RPs that communicate through our proxy, the proxy must request the total set of permissions that will be needed by the secure RPs. The proxy stores this confidential information inside its own database on the proxy server side.

Since the communication with the legacy IdP takes place using an insecure SSO implementation, the process should ideally be done in private browsing mode on a secure machine with no other web sites open. This is to prevent details such as the application ID of the proxy from being leaked to potential attackers.

Communication with the secure RP. After the initial setup, the secure RP communicates only with our proxy, instead of the legacy IdP. Communication between the RP and our proxy is done over the secure channel, as discussed in Section 4.1.3.1. We first authenticate the user, and then, in the case of an OAuth-like protocol, generate our own token for the RP with a corresponding subset of the privileges granted to us by the legacy IdP. In the case of an OpenID-like protocol, the proxy can simply forward the authentication proof directly to the RP.

Fetching the User's Information. In an OAuth-like protocol, when the RP asks for the user's information using the token issued by us, the proxy sends a request for this information to the legacy IdP with the token issued by that legacy IdP. When the legacy IdP returns the requested information, the proxy forwards the information to the RP.

4.1.4. Implementation

In this section, we present the implementation details of our system. First, in Section 4.1.4.1, we introduce the implementation of our clean-slate design. We implemented the authenticated, bi-directional, secure channel as a layer on top of the existing *postMessage* channel available in modern browsers, and then built our SSO protocol by leveraging this channel. Next, in Section 4.1.4.2, we discuss a prototype implementation of our proxy design.

4.1.4.1. IdP Deployment. Our implementation of the clean-slate design is very lightweight, consisting of only 252 lines of JavaScript code (excluding external libraries), 264 lines of HTML code, and 243 lines of PHP code. The bi-directional secure channel with authentication is implemented in JavaScript; the RP-side and IdP-side code is written in HTML and PHP.

Our JavaScript implementation of the secure channel uses two external libraries, the JavaScript Cryptography Toolkit [**22**] and the Stanford JavaScript Crypto Library [**45**], for cryptographic purposes. The former is used for public and private key generation and asymmetric encryption and

decryption; the latter is used for session key generation and symmetric encryption and decryption. The public and private keys in the implementation are 512 bits, and the session key is 128 bits. However, these are simply implementation details, and can be modified to address different security requirements. As mentioned before, we implemented the protocol by adding our security layer to the *postMessage* functionality available in modern browsers, which provides a reliable, but unsecured communication channel which guarantees in-order delivery of messages.

We use a socket pool for session tracking. For each incoming message, the system looks in the socket pool for a channel with the corresponding N_IdP and N_RP and, if one is found, fetches the socket and session key. Otherwise, a new socket is created, as described in Section 4.1.3.1, and stored in the socket pool.

To evaluate the practicality of our design, we created reference IdP and RP implementations in HTML and PHP. A login interface for the IdP prompts the user to input his username and password. The IdP's server-side code looks up the credentials in a database and, if the username and password match, it generates an access token and transfers the token to the IdP JavaScript. Since the authentication process involves multiple server-side pages, we load an iframe to embed the authentication process and communicate with the RP. This iframe talks to the IdP iframe that is responsible for authentication through another secure channel.

4.1.4.2. Proxy RP Deployment. To demonstrate the feasibility of our proxy design, we implemented a prototype RP that authenticates against Facebook through our proxy. The proxy process works as follows: first, a user who wants to authenticate via SSO through our proxy needs to register on our site. We enable the Facebook developer account for that user and let the user register a Facebook application under his account. This process is manual, since it involves solving a simple captcha and acknowledging the terms of service. The user then provides the proxy's application's ID and secret key, so that the proxy can act as the application. While technically not necessary, we

use the user's account to register an application because it lets the user maintain full control over the application, and the security is tied to the user's account. The whole process is done only once per user, as would happen when installing a normal application. A similar system can be implemented for Google's SSO implementation, which also adopts a user application key and secret to authenticate an application.

The user will then grant the proxy permissions to access his/her user data. As mentioned previously, these permissions are a superset of those that the proxy can provide to an RP. Notice that the aforementioned process should be done in a secure environment, such as private browsing with no other web sites open, to minimize the risk of leaking information to third parties.

Afterwards, when a user visits an RP supported by our proxy, the SSO process will be the same as with clean-slate design in the IdP deployment. We use Facebook's SDK [**17**], safely isolated from the RP, on the server-side to fetch the user's data and forward them to the RP.

There are several implementation concerns that arise in relation to the proxy prototype implementation.

Secrecy of the legacy IdP application ID. To guarantee the security of the proxy design, the application ID used on the legacy IdP should be secret. In the case of the legacy IdP being Facebook Connect, we take two measures to ensure this secrecy. First, we register a unique Facebook application ID for each user. This is done so that an attacker cannot learn the application ID by using the proxy service himself. Second, certain legacy IdPs, such as Facebook, provide a sandbox mode [**37**] for developer use. This sandbox mode isolates the application from all users except for the developer. In our case, the "developer" is the user of the proxy implementation, and this functionality allows us to keep the application ID secret.

Legacy IdP Terms of Service. When implementing the proxy design, care must be taken to avoid violating the legacy IdP terms of service. We have reviewed Facebook's TOS and determined

that encouraging every user to enable Facebook Developer Mode is acceptable. Additionally, we strove to make the legacy IdP portion of the proxy lightweight to avoid putting any significant load on Facebook's infrastructure.

Legacy IdP access token expiration. Tokens acquired from legacy IdPs, such as Facebook Connect, have an expiration time past which they cease granting access to a user's data. For Facebook Connect, this expiration is 60 days from the token's issuance time. After these 60 days, a reacquisition of the token by our proxy has to occur. This requires the user to redo the initial setup process, which should be done in a private browsing mode to guarantee security. Tokens granted by the proxy to the RP do not expire in the current implementation.

4.1.5. Evaluation

We first formally verify the channel in Section 4.1.5.1. Following, we empirically examine existing vulnerabilities in Section 4.1.5.2. Finally, we study the performance of our implementation in Section 4.1.5.3 to show that our protocol incurrs only a reasonable overhead for the security guarantees it provides.

4.1.5.1. Formal Protocol Verification. We verify the correctness of our protocol with ProVerif [**36, 71**], an automatic cryptographic protocol verifier using the Dolev-Yao model [**90**]. ProVerif can prove properties including secrecy, authentication, strong secrecy, and equivalences of a protocol. For our formal verification, we require only the former two properties.

In the following two sections, we model the bi-directional secure channel first, and model the SSO protocol built upon the channel second.

Channel Verification. We model the channel and the attacker, and then verify the channel using the attacker model.

Channel Modeling. We model our secure channel by using two processes and one free channel according to our threat model (c.f., Section 4.1.1). These two processes model the client-side and the server-side, respectively. The free channel, exposed to the attacker by definition, is used to communicate between the client and the server.

During the first step, the RP sends its own public key to the IdP. We generate private and public key pair, and then send the public key to the IdP. In the second step, the server verifies the origin (that is defined as a bitstring transmitted together with the public key in the first step). This is a hidden assumption because browsers always send the web origin as part of a message send through the postMessage channel. If the origin matches, in (3), we generate a symmetric session key. Next, at the IdP side, a method *aenc* is used to encrypt the session key and generate N_IdP. After receiving the encrypted session key, RP uses a method *adec* to decrypt the message in (4). Following, it encrypts a generated N_RP via a method *senc* with the session key. In the end, the IdP can send messages securely by using *senc* and appending N_IdP and N_RP.

Attacker Modeling. An attacker, in the context of a secure channel, is both an active and passive adversary who is capable of sending messages to either party and capable of eavesdropping on messages send across the channel. In ProVerif, a free channel has such properties and we adopted it here.

Results. After modeling the channel and attacker, we query ProVerif whether the attacker can obtain the plaintext of an encrypted message sent over the secure channel, and ProfVerif informs us that an attacker is unable to read the original, unencrypted message. To manifest our trust in this conclusion, we introduce a vulnerability into the protocol and investigate if ProVerif can produce an attack scenario in which the adversary is able to read the message in the clear, as we would expect it to do.

To introduce the vulnerability, we simply removed the origin check, performed by the IdP, of the RP. Once removed, ProVerif produces a counterexample. The produced attack works as follows: a malicious RP directly sends its own PK_RP to the IdP, and, in turn, acquires the session key SK and channel number N_IdP. Since the IdP does not verify the RP's identity, the malicious RP can impersonate the legitimate RP and talk to the IdP as if it is the legitimate RP.

SSO Protocol Verification. Again, we first model the SSO protocol and the attacker. Finally, we verify the SSO protocol based on our threat model.

SSO Protocol Modeling. First, we model the SSO protocol with two secure channels and three processes. The three processes represent the client-side RP, the client-side IdP and the server-side IdP. The client-side RP and the client-side IdP communicate with each other over the secure channel (as modeled in Section 4.1.5.1). The client-side IdP and the server-side IdP communicate through HTTPS, a well-known secure channel.

Attacker Modeling. In respect to our threat-model, in-scope attackers are network attackers and web attackers as defined by Akhawe et al [**57**]. A network attacker is a passive adversary capable of eavesdropping on the network traffic, while a web attacker can also control malicious web sites and clients.

Results. Both channels are verified as secure with properly authenticated peers. Because of this, an attacker cannot acquire any useful information from either channel. Like we expect, ProVerif simply confirms this result.

Proxy Design Verification. We first model the proxy and the attacker. We then verify the proxy using our attacker model.

Proxy Design Modeling. We model the proxy design with one secure channel, one insecure channel, and three processes. These three processes correspond to the RP, our proxy (the new,

secure, fourth-party IdP), and the real, legacy IdP. The channel between the RP and our proxy is secure, and the channel between our proxy and the legacy IdP is insecure.

Attacker Modeling. The attacker model follows our attacker model definition from the Channel Verification of Section 4.1.5.1.

Results. Since all communication of our proxy and the real IdP is over an insecure channel, a malicious RP can intercept any ongoing communication. Unsurprisingly, ProVerif yields that the information transmitted over the channel between our proxy and the real IdP can be obtained by an attacker. Thus, as discussed in Section 4.1.3.2, it is crucial that we need to keep the identity of our proxy at the real IdP confidential (to prevent impersonation attacks on the proxy) and that we perform the initial setup, during which communication between the proxy and the real IdP occurs, in a secure environment, such as in private browsing mode with no other web sites open to which information of the proxy might leak.

4.1.5.2. Security Analysis. In this section, we study several existing RP impersonation vulnerabilities [**155**] and show how our design effectively mitigates such vulnerabilities.

Facebook and NYTimes. The first vulnerability we discuss was found to be in the interaction between the NYTimes as the RP and Facebook as the IdP. In this vulnerability, involving Facebook and the NYTimes [**155**], a malicious RP first impersonates the NYTimes by spoofing its app_id. In turn, Facebook will continue to authenticate the user and generate an access token that is supposed to be delivered to the NYTimes. However, due to nature of cross-domain communication in Adobe Flash, *i.e.*, its "unpredictability" [**27**], the in-browser channel can be eavesdropped on and the access token is being leaked to the malicious RP, the actual initiator of the authentication request.

Mitigation. The vulnerability has two major components: a malicious RP impersonating the NYTimes and the insecure communication channel between the NYTimes and Facebook. In our protocol, both vulnerable parts are mitigated by design. First, we leverage web origins to verify

the identity of the RP, thus a malicious RP cannot impersonate the NYTimes. Second, the communication channel between the NYTimes and Facebook is a secure channel with authentication, therefore, even if a malicious RP would acquire the messages transmitted, it cannot easily decrypt and retrieve the clear message.

JanRain Wrapping GoogleID. A second interesting vulnerability was found in how JanRain wraps GoogleID [**155**]. Here, a malicious RP registers itself with JanRain. Initially, the malicious RP initiates the communication. Then, once JanRain redirects the communication to Google, the malicious RP impersonates the victim RP, but sets the return URL to its own URL. Since Google is using the HTTP redirection method described in Section 4.1.2.2 to communicate with the RP, confidential information will be leaked to the malicious RP.

Mitigation. In our design, when the RP talks to the IdP, all communication occurs over a bi-directional secure channel, meaning, when the real RP talks to the IdP, the IdP will respond to the original, legitimate RP rather than to a malicious RP. Thus, the vulnerability is mitigated by design.

Facebook and Zoho. Similar to the vulnerability found in how JanRain wraps GoogleID, a vulnerability in the use of HTTP redirection was discovered between Facebook and Zoho.com. To exploit the vulnerability in the interaction between Facebook and Zoho.com [**155**], after receiving the authorization code from the redirect_url of Zoho, if the attacker sends a request to Zoho and sets the service_url to evil.com, Zoho fails the SSO, but still redirects the user to evil.com.

Mitigation. The root cause of this vulnerability is very similar to 2). Since our protocol leverages a bi-directional secure channel, a return URL is not required in our design, thus preventing such vulnerability.

JanRain Wrapping Facebook. In this vulnerability, JanRain wraps Facebook as the IdP [**155**]. Here, *sears.com* incorrectly sets its whitelist to **.sears.com* rather than to the more restrictive *rp.sears.com*, thus exposing it to an attack similar to the *document.domain* attack by Singh et al. [**139**].

Mitigation. Since our design specifically uses web origins as the identity of the RP, the concept of a whitelist is not being used by us, thus preventing the misconfiguration of any such whitelist by design.

Facebook Legacy Canvas Auth. Lastly, a third vulnerability that allows IdP impersonation rather than RP impersonation was discovered in how Facebook is being used as the IdP. In this case, the vulnerability lies in in the fact that, for Facebook's legacy canvas authentication [**155**], the generated signature is not properly verified by a Facebook app (verified for *FarmVille.com*). Because of this, the RP renders itself vulnerable to IdP impersonation attacks, thus, effectively, it renders itself vulnerable to arbitrary user impersonation attacks, potentially even leaking private or confidential user data.

Mitigation. Since we transmit every message in an established secure channel that provides authentication already, it is not necessary for the RP to verify any signature itself. For this reason, it reduces the risk of missing or forgetting signature verification steps and preventing the impersonation attacks that might arise here.

4.1.5.3. Performance Analysis. Environment and Methodology. Our client-side experiment was performed on a 2.67GHz Intel(R) i7 CPU with four physical cores and 8GB of memory running Ubuntu 13.04 64-bit. The browser at the client-side was Firefox 22.0 32-bit. The RP server was deployed on a server running CentOS 64-bit with 2.50GHz Intel(R) Xeon(R) CPU with eight physical cores and 16GB memory. The IdP server was deployed on a server running CentOS 64-bit

Table 4.1. Breakdown of the authentication performance of our prototype implementation.

Operation	**Delay**
(1) Creating the Channel between RP and IdP	164±11ms
(2) Creating IdP Iframe	57±3ms
(3) Sending the First Message from RP to IdP	32±2ms
(4) Creating IdP Iframe for Authentication	57±3ms
(5) Creating the Second Channel inside IdP	165±11ms
(6) Authenticating the User	56±4ms
(7) Getting the User's Permission	57±3ms
(8) Sending the Token inside IdP Iframe	32±2ms
(9) Sending the Token to RP	33±2ms
Total	653±21ms

Note: Step (2), (4), (6) and (7) are extremely depended on the network latency.

with a 2.80GHz Intel(R) Xeon(R) CPU with four cores and 16GB memory. The average round-trip network latency between the client and the two servers was measured to be about 50ms.

To measure the delay of each step of the secure channel and an instantiation of our SSO protocol, we make use of the JavaScript primitive *Date()* and subtract the monitored value at start and end points to calculate the delay. Each experiment was repeated ten times to calculate the average delay and to account for outliers and deviations. In the following two sections, we measure the performance of the secure channel first, and the performance of our SSO prototype implementation after.

Performance of SSO Implementation. After measuring the performance of our bi-directional secure channel with authentication, we measure the performance of our prototype implementation of our SSO protocol. The results are shown in Table 4.1. They are divided into three categories: channel creation, network delay, and message passing. (1) and (5) belong to channel creation, which takes about 164ms. (2), (4), (6) and (7) are almost exclusively driven by the network delay. (2) and (4) are to fetch content from the IdP, and (6) and (7) are used to communicate with the IdP. In the end, (3), (8), and (9) are used to communicate between iframes within the browser through our established secure channel, which is about 32ms. Overall, the total overhead of our prototype

implementation is only 653ms, i.e., acceptable from a user's perspective given the strong security guarantees of our SSO protocol.

CHAPTER 5

Conclusion

In my thesis, I have proposes three fundamental questions of web security. **Future work** in Web security, in my opinion, turns on how to solve the three aforementioned fundamental questions, and how to apply techniques for solving these questions to real-world practical problems.

None of the three aforementioned questions are fully solved. A stronger principal creates higher overhead, and it is a tradeoff to find the balance between security and high performance. Similarly, if we define a finer-grained principal by putting less content into a principal, we need to introduce more communication between principals, i.e., additional overhead. On contrary, if we define a coarse-grained principal by putting more content into a principal, security may be compromised. Again, a balance point is necessary.

More importantly, these three fundamental problems do not stand alone by themselves, i.e., they interact greatly with the most cutting-edge Web-borne threats. I will illustrate the point from two possible directions: defending Java zero-day vulnerabilities and preventing third-party Web tracking.

Java has recently exhibited a huge number[1] of zero-day vulnerabilities, most of which belong to privilege escalation. A malicious Java program from the server can bypass the access control enforced by the sandbox of Java virtual machine (JVM) in the browser, and gain high privileges of the JVM. When applying the principal-based browser architecture to Java, a browser plug-in, we need to isolate the server-provided Java program with low privilege, and Java library with high

[1]It is reported that Java zero-day holes appear at a rate as high as one a day (`http://www.infoworld.com/t/java-programming/java-zero-day-holes-appearing-the-rate-of-one-day-213898`).

privilege, into different principals. Then, a communication channel will be created for the server-provided Java program to use the Java library. In sum, the principal-based browser architecture can also be adopted for JVM.

Web tracking attracts the public's attention, since privacy becomes an important issue for the Internet. Among those, third-party Web tracking is more important, because your browsing behavior is leaked to a third-party that you may not know. In one sentence, those third-party Web sites use cookies (or its counterparts) to fingerprint a user, and *referer*[2] header to record which Web site the user has visited. When applying the principal-based browser architecture to third-party Web tracking, we need to isolate a third-party web site with *referer* header as `a.com` and that with a *referer* header as `b.com`. Thus, the third-party Web site cannot connect the behaviors of that user when he visits both `a.com` and `b.com`.

[2]The misspelling of referer comes from RFC 1945 (`http://en.wikipedia.org/wiki/HTTP_referer`).

References

[1] http://www.browserauth.net/channel-bound-cookies.

[2] AD Safe. http://www.adsafe.org/.

[3] Alexa Top Websites. http://www.alexa.com/topsites.

[4] Apple hit by hackers who struck facebook. http://online.wsj.com/article/SB10001424127887324449104578314321123497696.html.

[5] CAPTCHA. http://en.wikipedia.org/wiki/CAPTCHA.

[6] Cisco ips signatures. http://tools.cisco.com/security/center/ipshome.x?i=62&shortna=CiscoIPSSignatures#CiscoIPSSignatures.

[7] Common vulnerabilities and exposures. http://cve.mitre.org/.

[8] Content Security Policy - Mozilla. http://people.mozilla.com/~bsterne/content-security-policy/index.html.

[9] Cross Site Request Forgery (CSRF) - OWASP.

[10] Cryptographically secure pseudo-random number generator. http://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator.

[11] CVE-2007-4139. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-4139.

[12] DOM-based XSS attack. https://www.owasp.org/index.php/DOM_Based_XSS.

[13] Doxygen. http://www.stack.nl/~dimitri/doxygen/.

[14] Elgg. http://www.elgg.org/.

[15] FBJS. http://wiki.developers.facebook.com/index.php/FBJS.

[16] A firefox pdf plug-in XSS vulnerability. http://lwn.net/Articles/216223/.

[17] Getting started with the facebook sdk for php. https://developers.facebook.com/docs/php/gettingstarted/.

[18] Google code home page of configurable origin policy. http://code.google.com/p/configurableoriginpolicy/.

[19] Google Friend Connect - Google. http://code.google.com/apis/friendconnect/.

[20] HTML5: A vocabulary and associated APIs for HTML and XHTML.

[21] JanRain Engage. http://janrain.com/products/engage/.

[22] JavaScript Cryptography Toolkit. http://ats.oka.nu/titaniumcore/js/crypto/readme.txt.

[23] Javascript game site. http://javascript.internet.com/games/.

[24] JavaScript reference. https://developer.mozilla.org/en/Core_Javascript_1.5_Reference.

[25] Javascript test cases. http://mxr.mozilla.org/mozilla/source/js/tests/ecma/.

[26] Javascript Yamanner worm. http://www.theregister.co.uk/2006/06/12/javscript_worm_targets_yahoo/.

[27] LocalConnection. http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/net/LocalConnection.html.

[28] Mcafee q1 2011 report ii. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2011.pdf.

[29] A modular universal xss worm. http://groups.google.com/group/ph4nt0m/msg/d75435c75fc6b81b?pli=1.

[30] Mozilla rejects native code approach of chrome's nacl. http://css.dzone.com/articles/mozilla-rejects-native-code.

[31] Myspace samy worm. http://namb.la/popular/tech.html.

[32] OpenID. http://openid.net/.

[33] Parsing time complexity. http://en.wikipedia.org/wiki/Top-down_parsing.

[34] Private Browsing - Firefox. http://support.mozilla.com/en-us/kb/private+browsing.

[35] Privoxy. www.privoxy.org.

[36] ProVerif: Cryptographic protocol verifier in the formal model. http://prosecco.gforge.inria.fr/personal/bblanche/proverif/.

[37] Sandbox mode of facebook application. https://developers.facebook.com/docs/ApplicationSecurity/.

[38] Secure electronic transaction. http://goo.gl/2SpMbF.

[39] Security assertion markup language. http://en.wikipedia.org/wiki/Security_Assertion_Markup_Language.

[40] Session Definition - Wikipedia. http://en.wikipedia.org/wiki/Session_(computer_science).

[41] Snort rules. http://www.snort.org/snort-rules/.

[42] Spaceflash worm on MySpace. http://news.cnet.com/Worm-lurks-behind-MySpace-profiles/2100-7349_3-6095533.html.

[43] Spidermonkey javascript engine. https://developer.mozilla.org/en-US/docs/SpiderMonkey.

[44] Symentec internet threat report. http://www.symantec.com/threatreport/.

[45] The Stanford Javascript Crypto Library. http://crypto.stanford.edu/sjcl/.

[46] W3C Working Draft - Cross-Origin Resource Sharing.

[47] Webkit source codes. http://webkit.org/building/checkout.html.

[48] White hat report. https://www.whitehatsec.com/assets/WPStatsreport_100107.pdf.

[49] Wordpress. http://wordpress.org/.

[50] XDomainRequest - IE8. http://msdn.microsoft.com/en-us/library/dd573303%28VS.85%29.aspx.

[51] XSS Cheat Sheet. http://ha.ckers.org/xss.html.

[52] XSS definition and classification in Wikipedia. http://en.wikipedia.org/wiki/Cross-site_scripting.

[53] XSS worm on Renren social network, 2009. http://issmall.isgreat.org/blog/archives/2.

[54] Boonana Java worm, 2010. http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Worm%3AJava%2FBoonana.

[55] Facebook hit by XSS worm, 2011. http://news.softpedia.com/news/Facebook-Hit-by-XSS-Worm-192045.shtml.

[56] ACKER, S. V., RYCK, P. D., DESMET, L., PIESSENS, F., AND JOOSEN, W. Webjail: Least-privilege integration of third-party components in web mashups. In *Annual Computer Security Applications Conference (ACSAC)* (2011).

[57] AKHAWE, D., BARTH, A., LAM, P. E., MITCHELL, J. C., AND SONG, D. Towards a formal foundation of web security. In *CSF: the Computer Security Foundations Symposium* (2010).

[58] BAI, G., LEI, J., MENG, G., VENKATRAMAN, S. S., SAXENA, P., SUN, J., LIU, Y., AND DONG, J. S. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS* (2013).

[59] BALZAROTTI, D., COVA, M., FELMETSGER, V., JOVANOVIC, N., KIRDA, E., KRUEGEL, C., AND VIGNA, G. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 387–401.

[60] BALZAROTTI, D., COVA, M., FELMETSGER, V. V., AND VIGNA, G. Multi-module vulnerability analysis of web-based applications. In *CCS: Conference on Computer and Communication Security* (2007).

[61] BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *OSDI: Symposium on Operating Systems Design and Implementation* (1999).

[62] BARTH, A., CABALLERO, J., AND SONG, D. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *SP:IEEE Symposium on Security and Privacy* (2009).

[63] BARTH, A., JACKSON, C., AND HICKSON, I. The HTTP Origin Header - IETF Draft.

[64] BARTH, A., JACKSON, C., AND LI, W. Attacks on javascript mashup communication. In *W2SP: Web 2.0 Security and Privacy* (2009).

[65] BARTH, A., JACKSON, C., AND MITCHELL, J. Robust defenses for cross-site request forgery. In *CCS* (2008).

[66] BARTH, A., JACKSON, C., AND MITCHELL, J. C. Securing frame communication in browsers. In *USENIX Security Symposium* (2008).

[67] BARTH, A., WEINBERGER, J., AND SONG, D. Cross-origin javascript capability leaks: Detection, exploitation, and defense. In *USENIX Security Symposium* (2009).

[68] BATES, D., BARTH, A., AND JACKSON, C. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th International Conference on World Wide Web* (New York, NY, USA, 2010), WWW '10, ACM, pp. 91–100.

[69] BHARGAVAN, K., DELIGNAT-LAVAUD, A., AND MAFFEIS, S. Language-based defenses against untrusted browser origins. In *USENIX Security Symposium* (2013).

[70] BISHT, P., AND VENKATAKRISHNAN, V. N. XSS-GUARD: Precise dynamic prevention of cross-site scripting attacks. In *Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (Berlin, Heidelberg, 2008), DIMVA '08, Springer-Verlag, pp. 23–43.

[71] BLANCHET, B. An efficient cryptographic protocol verifier based on prolog rules. In *CSFW* (2001).

[72] BORTZ, A., BARTH, A., AND CZESKIS, A. Origin cookies: Session integrity for web applications. In *W2SP: Web 2.0 Security and Privacy* (2011).

[73] BRUMLEY, D., NEWSOME, J., SONG, D., WANG, H., AND JHA, S. Towards automatic generation of vulnerability-based signatures. In *SP: the 2006 IEEE Symposium on Security and Privacy* (2006).

[74] CANALI, D., COVA, M., VIGNA, G., AND KRUEGEL, C. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *WWW: the 20th international conference on World wide web* (2011).

[75] CAO, Y., LI, Z., RASTOGI, V., AND CHEN, Y. Virtual browser: a web-level sandbox to secure third-party javascript without sacrificing functionality (poster paper). In *ACM conference on Computer and communications security (CCS)* (2010).

[76] CAO, Y., LI, Z., RASTOGI, V., WEN, X., AND CHEN, Y. Virtual browser: a virtualized browser to sandbox third-party javascripts with enhanced security. In *ACM ASIACCS* (2012).

[77] CAO, Y., PAN, X., CHEN, Y., AND ZHUGE, J. JShield: Towards complete de-obfuscation and real-time detection of complex drive-by download attacks (under review).

[78] CAO, Y., PAN, X., CHEN, Y., AND ZHUGE, J. Jshield: Towards real-time and vulnerability-based detection of polluted drive-by download attacks. In *Technical Report (Northwestern University)* (2013). `http://www.cs.northwestern.edu/~yca179/JShield/JShield.pdf`.

[79] CAO, Y., RASTOGI, V., LI, Z., CHEN, Y., AND MOSHCHUK, A. Redefining web browser principals with a configurable origin policy. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN - DCCS)* (2013).

[80] CAO, Y., SHOSHITAISHVILI, Y., BORGOLTE, K., KRUEGEL, C., VIGNA, G., AND CHEN, Y. Protecting web single sign-on against relying party impersonation attacks through a bi-directional secure channel with authentication (under review).

[81] CAO, Y., YEGNESWARAN, V., PORRAS, P., AND CHEN, Y. Pathcutter: Severing the self-propagation path of xss javascript worms in social web networks. In *Network and Distributed System Security Symposium (NDSS)* (2012).

[82] CHEN, E. Y., BAU, J., REIS, C., BARTH, A., AND JACKSON, C. App isolation: get the security of multiple browsers with just one. In *CCS: conference on Computer and communications security* (2011).

[83] CHONG, S., VIKRAM, K., AND MYERS, A. C. SIF: enforcing confidentiality and integrity in web applications. In *USENIX Security Symposium* (2007).

[84] COVA, M., KRUEGEL, C., AND VIGNA, G. Detection and analysis of drive-by-download attacks and malicious javascript code. In *WWW* (2010).

[85] COX, R. S., GRIBBLE, S. D., LEVY, H. M., AND HANSEN, J. G. A safety-oriented platform for web applications. In *SP: IEEE Symposium on Security and Privacy* (2006).

[86] CRITES, S., HSU, F., AND CHEN, H. OMash: Enabling secure web mashups via object abstractions. In *CCS: Conference on Computer and Communication Security* (2008).

[87] CURTSINGER, C., LIVSHITS, B., ZORN, B., AND SEIFERT, C. Zozzle: fast and precise in-browser javascript malware detection. In *the 20th USENIX conference on Security* (2011).

[88] DABIRSIAGHI, A. Building and stopping next generation XSS worms. In *3rd International OWASP Symposium on Web Application Security* (2008).

[89] DE KEUKELAERE, F., BHOLA, S., STEINER, M., CHARI, S., AND YOSHIHAMA, S. SMash: Secure component model for cross-domain mashups on unmodified browsers. In *WWW: Conference on World Wide Web* (2008).

[90] DOLEV, D., AND YAO, A. C. On the security of public key protocols. Tech. rep., Stanford, CA, USA, 1981.

[91] DONG, X., TRAN, M., LIANG, Z., AND JIANG, X. Adsentry: Comprehensive and flexible confinement of javascript-based advertisements. In *Annual Computer Security Applications Conference (ACSAC)* (2011).

[92] FACEBOOK. Facebook connect. `http://developers.facebook.com/blog/post/108/`.

[93] FINIFTER, M., WEINBERGER, J., AND BARTH, A. Preventing capability leaks in secure javascript subsets. In *NDSS: Network and Distributed System Security Symposium* (2010).

[94] GOOGLE. Google Caja. `http://code.google.com/p/google-caja/`.

[95] GOOGLE. Using multiple accounts simultaneously. `http://www.google.com/support/accounts/bin/topic.py?hl=en&topic=28776`.

[96] GRIER, C., TANG, S., AND KING, S. T. Secure web browsing with the OP web browser. In *SP: IEEE Symposium on Security and Privacy* (2008).

[97] GUARNIERI, S., AND LIVSHITS, B. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *18th USENIX Security Symposium* (2009).

[98] HANNA, S., SHIN, R., AKHAWE, D., SAXENA, P., BOEHM, A., AND SONG, D. The emperor's new APIs: On the (in)secure usage of new client-side primitives. In *W2SP: Web 2.0 Security and Privacy* (2010).

[99] HOFFMAN, B. Analysis of web application worms and viruses. In *Blackhat 06*. `http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Hoffman/BH-Fed-06-Hoffman-up.pdf`.

[100] HUANG, L.-S., WEINBERG, Z., EVANS, C., AND JACKSON, C. Protecting browsers from cross-origin CSS attacks. In *CCS: Conference on Computer and Communications Security* (2010).

[101] HUANG, Y.-W., YU, F., HANG, C., TSAI, C.-H., LEE, D.-T., AND KUO, S.-Y. Securing web application code by static analysis and runtime protection. In *WWW: Conference on World Wide Web* (2004).

[102] INTERNATIONAL SECURE SYSTEMS LAB. Wepawet. `http://wepawet.iseclab.org/`.

[103] IOANNIDIS, S., AND BELLOVIN, S. M. Building a secure web browser. In *USENIX Annual Technical Conference* (2001).

[104] IOANNIDIS, S., BELLOVIN, S. M., AND SMITH, J. M. Sub-operating systems: a new approach to application security. In *EW: ACM SIGOPS European workshop* (2002).

[105] JACKSON, C., AND BARTH, A. Beware of finer-grained origins. In *W2SP: Web 2.0 Security and Privacy* (2008).

[106] JACKSON, D. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[107] JAYARAMAN, K., DU, W., RAJAGOPALAN, B., AND CHAPIN, S. J. Escudo: A fine-grained protection model for web browsers. In *International Conference on Distributed Computing Systems - ICDCS* (2010).

[108] JENSEN, S. H., MØLLER, A., AND THIEMANN, P. Type analysis for javascript. In *SAS: the International Symposium on Static Analysis* (2009).

[109] JIM, T., SWAMY, N., AND HICKS, M. Defeating script injection attacks with browser-enforced embedded policies. In *Proceedings of the 16th international conference on World Wide Web* (New York, NY, USA, 2007), WWW '07, ACM, pp. 601–610.

[110] JOVANOVIC, N., KRUEGEL, C., AND KIRDA, E. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). In *SP: IEEE Symposium on Security and Privacy* (2006).

[111] KARLOF, C., SHANKAR, U., TYGAR, J. D., AND WAGNER, D. Dynamic pharming attacks and locked same-origin policies for web browsers. In *CCS: Conference on Computer and Communication Security* (2007).

[112] KIKUCHI, H., YU, D., CHANDER, A., INAMURA, H., AND SERIKOV, I. Javascript instrumentation in practice. In *APLAS: Asian Symposium on Programming Languages and Systems* (2008).

[113] KIRDA, E., KRUEGEL, C., VIGNA, G., AND JOVANOVIC, N. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC: ACM symposium on Applied computing* (2006).

[114] LI, Z., TANG, Y., CAO, Y., RASTOGI, V., CHEN, Y., LIU, B., AND SBISA, C. Webshield: Enabling various web defense techniques without client side modifications. In *Network and Distributed System Security Symposium (NDSS)* (2011).

[115] LI, Z., XIA, G., GAO, H., YI, T., CHEN, Y., LIU, B., JIANG, J., AND LV, Y. Netshield: Massive semantics-based vulnerability signature matching for high-speed networks. In *SIGCOMM* (2010).

[116] LIVSHITS, B., AND CUI, W. Spectator: detection and containment of javascript worms. In *ATC: USENIX Annual Technical Conference* (2008).

[117] LIVSHITS, V. B., AND LAM, M. S. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14* (Berkeley, CA, USA, 2005), USENIX Association, pp. 18–18.

[118] LU, X., ZHUGE, J., WANG, R., CAO, Y., AND CHEN, Y. De-obfuscation and detection of malicious pdf files with high accuracy. In *Hawaii International Conference on System Sciences* (2013).

[119] LU, X., ZHUGE, J., WANG, R., CAO, Y., AND CHEN, Y. De-obfuscation and detection of malicious pdf files with high accuracy. In *HICSS* (2013).

[120] LUO, T., AND DU, W. Contego: Capability-based access control for web browsers - (short paper). In *Trust and Trustworthy Computing - 4th International Conference - TRUST* (2011).

[121] MAFFEIS, S., MITCHELL, J., AND TALY, A. Run-time enforcement of secure javascript subsets. In *W2SP: Web 2.0 Security and Privacy* (2009).

[122] MAGENTO INC. Magento. `http://www.magentocommerce.com/`.

[123] MARTIN, M., AND LAM, M. S. Automatic generation of XSS and SQL injection attacks with goal-directed model checking. In *Proceedings of the 17th Conference on Security Symposium* (Berkeley, CA, USA, 2008), USENIX Association, pp. 31–43.

[124] MEYEROVICH, L., FELT, A. P., AND MILLER, M. Object views: Fine-grained sharing in browsers. In *WWW: Conference on World Wide Web* (2010).

[125] MEYEROVICH, L., AND LIVSHITS, B. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy* (2010).

[126] MICROSOFT LIVE LABS. WebSandbox. http://websandbox.livelabs.com/.

[127] MOZILLA. Narcissus javascript engine. http://mxr.mozilla.org/mozilla/source/js/narcissus/.

[128] NADJI, Y., SAXENA, P., AND SONG, D. Document structure integrity: A robust basis for cross-site scripting defense. In *Network and Distributed System Security Symposium* (2009).

[129] ODA, T., WURSTER, G., VAN OORSCHOT, P. C., AND SOMAYAJI, A. SOMA: Mutual approval for included content in web pages. In *CCS: Conference on Computer and Communications Security* (2008).

[130] PERDISCI, R., DAGON, D., LEE, W., FOGLA, P., AND SHARIF, M. Misleading worm signature generators using deliberate noise injection. In *In Proceedings of the 2006 IEEE Symposium on Security and Privacy* (2006).

[131] POLITZ, J. G., ELIOPOULOS, S. A., GUHA, A., AND KRISHNAMURTHI, S. ADsafety: type-based verification of JavaScript sandboxing. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 12–12.

[132] REIS, C., DUNAGAN, J., WANG, H. J., DUBROVSKY, O., AND ESMEIR, S. Browsershield: vulnerability-driven filtering of dynamic html. In *OSDI* (2006).

[133] REIS, C., AND GRIBBLE, S. D. Isolating web programs in modern browser architectures. In *EuroSys* (2009).

[134] REIS, C., GRIBBLE, S. D., AND LEVY, H. M. Abstract architectural principles for safe web programs. In *HotNets: The Workshop on Hot Topics in Networks* (2007).

[135] RESIG, J. HTMLParser. http://ejohn.org/blog/pure-javascript-html-parser/.

[136] RIECK, K., KRUEGER, T., AND DEWALD, A. Cujo: efficient detection and prevention of drive-by-download attacks. In *ACSAC* (2010).

[137] SAXENA, P., AKHAWE, D., HANNA, S., MAO, F., MCCAMANT, S., AND SONG, D. A symbolic execution framework for javascript. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2010), SP '10, IEEE Computer Society, pp. 513–528.

[138] SENOCULAR. CSSParser. http://www.senocular.com/index.php?id=1.289.

[139] SINGH, K., MOSHCHUK, A., WANG, H., AND LEE, W. On the Incoherencies in Web Browser Access Control Policies. In *SP: IEEE Symposium on Security and Privacy* (2010).

[140] SON, S., AND SHMATIKOV, V. The Postman Always Rings Twice: Attacking and Defending postMessage in HTML5 Websites. In *NDSS* (2013).

[141] SONG, C., ZHUGE, J., HAN, X., AND YE, Z. Preventing drive-by download via inter-module communication monitoring. In *ASIACCS* (2010).

[142] SOTIROV, A. Blackbox reversing of xss filters. *RECON* (2008).

[143] SUN, F., XU, L., AND SU, Z. Client-side detection of XSS worms by monitoring payload propagation. In *ESORICS* (2009), M. Backes and P. Ning, Eds., vol. 5789 of *Lecture Notes in Computer Science*, Springer, pp. 539–554.

[144] SUN, S.-T., AND BEZNOSOV, K. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *CCS* (2012).

[145] TANG, S., GRIER, C., ACIICMEZ, O., AND KING, S. T. Alhambra: a system for creating, enforcing, and testing browser security policies. In *WWW: Conference on World Wide Web* (2010).

[146] TANG, S., MAI, H., AND KING, S. T. Trust and protection in the illinois browser operating system. In *OSDI* (2010).

[147] TASSANAVIBOON, A., AND GONG, G. Oauth and abe based authorization in semi-trusted cloud computing: aauth. In *DataCloud-SC: the international workshop on Data intensive computing in the clouds* (2011).

[148] TER LOUW, M., GANESH, K. T., AND VENKATAKRISHNAN, V. Adjail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *19th USENIX Security Symposium* (2010).

[149] TER LOUW, M., AND VENKATAKRISHNAN, V. Blueprint: Precise browser-neutral prevention of cross-site scripting attacks. In *30th IEEE Symposium on Security and Privacy* (2009).

[150] URUEÑA, M., MUÑOZ, A., AND LARRABEITI, D. Analysis of privacy vulnerabilities in single sign-on mechanisms for multimedia websites. *Multimedia Tools and Applications* (2012).

[151] WANG, H. J., FAN, X., JACKSON, C., AND HOWELL, J. Protection and communication abstractions for web browsers in MashupOS. In *SOSP: ACM Symposium on Operating Systems Principles* (2007).

[152] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal os construction of the gazelle web browser. In *the conference on USENIX security symposium* (2009).

[153] WANG, H. J., GRIER, C., MOSHCHUK, A., KING, S. T., CHOUDHURY, P., AND VENTER, H. The multi-principal OS construction of the gazelle web browser. In *18th Usenix Security Symposium* (2009).

[154] WANG, H. J., GUO, C., SIMON, D. R., AND ZUGENMAIER, A. Shield: vulnerability-driven network filters for preventing known vulnerability exploits. In *SIGCOMM* (2004).

[155] WANG, R., CHEN, S., AND WANG, X. Signing me onto your accounts through facebook and google: a traffic-guided security study of commercially deployed single-sign-on web services. In *IEEE Symposium on Security and Privacy* (2012).

[156] WANG, R., ZHOU, Y., CHEN, S., QADEER, S., EVANS, D., AND GUREVICH, Y. Explicating sdks: Uncovering assumptions underlying secure authentication and authorization. In *USENIX Security Symposium* (2013).

[157] XIE, Y., AND AIKEN, A. Static detection of security vulnerabilities in scripting languages. In *USENIX Security Symposium* (2006).

[158] XING, L., CHEN, Y., WANG, X., AND CHEN, S. InteGuard: Toward Automatic Protection of Third-party web service integrations. In *NDSS* (2013).

[159] XU, W., ZHANG, F., AND ZHU, S. Toward worm detection in online social networks. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 11–20.

[160] YANG, E. Z., STEFAN, D., MITCHELL, J., MAZIERES, D., MARCHENKO, P., AND KARP, B. Toward principled browser security. In *HotOS* (2013).

[161] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *SP: IEEE Symposium on Security and Privacy* (2009).

[162] YU, D., CHANDER, A., ISLAM, N., AND SERIKOV, I. Javascript instrumentation for browser security. In *POPL: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2007).

[163] YUE, C., AND WANG, H. Characterizing insecure javascript practices on the web. In *WWW: Conference on the World wide web* (2009).