JASPIN: JAVASCRIPT BASED ANOMALY DETECTION OF CROSS-SITE SCRIPTING ATTACKS

by
Preeti Raman

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

MASTER OF COMPUTER SCIENCE

School of Computer Science

at

CARLETON UNIVERSITY

Ottawa, Ontario
September, 2008

Your file Votre référence
ISBN: 978-0-494-44111-4
Our file Notre référence
ISBN: 978-0-494-44111-4

Table of Contents

List of Tables

List of Figures

Abstract

The increasing use of sophisticated JavaScript in web applications has led to the widespread exploitation of cross-site scripting (XSS) flaws. We present an anomaly detection-based approach for detecting cross-site attacks. JaSPIn is based on the observation that the patterns of JavaScript methods invoked by web sites is extremely consistent, even for complex AJAX-driven applications. Thus, web page behaviorial profiles can be generated by recording the methods executed when legitimate content is displayed. These profiles can then be used to constrain JavaScript behavior so that XSS attacks cannot succeed. In experiments using JaSPIn implemented in Mozilla Firefox, we found that it generates stable website JavaScript profiles in few visits, and was subsequently able to detect a range of XSS attacks.

Acknowledgements

I wish to express my sincere thanks and appreciation to my advisor, Dr. Anil Somayaji for his attention, guidance, insight, patience and support during this research and the preparation of this thesis. In addition, special thanks are due to Dr. Paul Van Oorshot and Dr. Liam Peyton for their constructive comments and suggestions to the initial versions of this writing. I also would like to thank Dr. Evangelos Kranakis for serving on my committee and Dr David Mould for chairing my defense. Many thanks to the researchers at the Carleton Computer Security Lab for their interesting comments and suggestions. I also would like to thank RIM for funding this work.

Finally, I would like to thank my parents for their constant encouragement, my brother, Vinay for giving me company on those long nights and Aditya and Nemo for always putting a smile back on my face. I cannot finish without saying how grateful I am to have my unbelievably supportive and loving husband Abhay in my life.

Chapter 1

Introduction

The World Wide Web is an important medium of communication today. It has evolved from a static medium with user interaction limited to navigation between web pages to a highly interactive medium serving up personalized content. Web users can email, search, blog, chat, stream videos, play online games and shop on the Internet. Web applications are even used in security-critical environments, such as medical, financial, and military systems. This popularity of the web is enabled by various technologies that have evolved along with the growth of the World Wide Web. One such technology, JavaScript [Fla98], is widely used to enable the interactivity of web pages. A 2006 United Nations global study showed that 73 percent of websites surveyed relied on JavaScript for important functionality [Nom06].

JavaScript code is downloaded to the client computer as part of a web page and executed automatically in the browser by an embedded interpreter. Such automatically executed code could be malicious and potentially harm the user's environment. To protect the client from unauthorized access, a number of safeguards are built into the browser to prevent JavaScript running in the browser from gaining access to local machines. This mechanism is collectively referred to as the sandbox, a restricted area in which JavaScript can execute. Client side JavaScript runs in its own sandbox, and does not have access to anything outside the sandbox by design. Also, JavaScript programs downloaded from different sites are protected from each other using the same origin policy [Rud01] which permits code access only to resources associated with its origin site.

An issue with current JavaScript security mechanisms is that scripts may adhere to the same origin policy and be confined to their respective sandbox, but still be able to launch an attack. This is possible when an attacker is able to trick the user into downloading malicious JavaScript code from a web site trusted by the user. Such

JavaScript based attacks across sites are explained in the next section.

1.1 JavaScript Attacks

In the past couple of years, multiple attacks have been launched against websites using JavaScript. The majority of these attacks use some form of cross site scripting (XSS) or cross site request forgery (CSRF). Cross-site scripting (XSS) is an attack against web applications in which scripting code is typically injected into the output of an application that is then sent to a user's web browser [VNJ+07]. This scripting code executes in the browser with the privileges of the originating site, there by circumventing the same origin policy of the web browser. Malicious JavaScript code can then steal confidential user data, redirect the user to a different site, gain access to local files, perform tasks with system privileges and even launch other attacks such as SQL injection [SW06] and phishing [DTH06]. These attacks can cause damage both to the end user and the web application host. Consider the case of a malicious script stealing cookies stored by a banking site from a user's browser. The attacker can now not only cause financial damage to the user whose cookies were stolen, but also tarnish the reputation of the bank, causing loss of money, time and customers to the bank.

Cross-site Request Forgery (CSRF) [Law07] is an attack in which unauthorized commands are transmitted to a web site from a trusted user. While XSS exploits the trust a user has for a particular site, CSRF exploits the trust that a site has for a particular user.

One of the reasons cross-site attacks are so prevalent is the enormous number of freely available vulnerable web applications. Support from browsers in executing the attacker's JavaScript makes propagation of the attack easy and fast. JavaScript vulnerabilities leading to cross site scripting attacks are the most common of all publicly reported security vulnerabilities since 2005. [PFH03]. More than 25,000 sites have been discovered vulnerable to XSS attacks [FP07]. This is only a partial account of the number of vulnerable sites, since there are a number of web-based applications that have been developed internally by companies to provide customized services that could be vulnerable to both XSS and CSRF attacks.

The most desirable way to prevent and fix XSS vulnerabilities is by fixing the vulnerable code or to perform proper input validation. Of course, fixing the code is not the easiest or even the most practical solution in many cases, given that web pages today are dynamic and have multiple sources for advertisements, video, blog trackbacks, polls and other features [OSW08]. Further, it takes time to develop and test a patch for a newly discovered vulnerability, and it takes more time to patch third party software used in a given web application. Many researchers have proposed server side solutions to block XSS attacks without having to regularly fix the application code [SS02b], [Inc02], [KKP03], [HYH+04], [Min05], [XBS06], [JKK06], [Cor06]. These approaches however leave the user completely vulnerable if the website has not implemented any such measures. A complementary approach is to implement attack detection mechanisms at the client. Browser based defenses require methods to differentiate malicious JavaScript code downloaded from a trusted web site from normal JavaScript code, or techniques to mitigate the impact of cross-site scripting attacks. Various signature, policy and client side proxy based approaches [IEKY04], [HV05], [KKVJ06], [YCIS07], [VNJ+07], [JSH07], [UELX07], [JB07] have been proposed to detect cross-site attacks at the client. However, filter evasion techniques allow XSS attacks to avade many of these systems.

Robust JavaScript level attack detection is possible in the browser because any active content is executed only when the browser parses the content as a script. If an attack bypasses the JavaScript interpreter, no attack occurs. Our current work is based on the idea that when parsing scripts, if the browser is able to determine which scripts are out of the ordinary and possibly malicious, it could detect and even prevent cross site scripting attacks by disabling the execution of the script.

1.2 JaSPIn

We propose the first client side browser based defense, JaSPIn (JavaScript Profile Inferer), that can detect a wide range of XSS attacks, including zero-day attacks, using automatically generated web page profiles without using a whitelist. Further, our proposed solution is immune to common filter evasion techniques and does not require any changes to web applications.

Specifically, JaSPIn detects XSS attacks by building fine-grained JavaScript profiles for a given website. We have found that the pattern of JavaScript methods invoked by a web page is very consistent, even for complex AJAX-driven websites. Thus, specific JavaScript security profiles can be generated by recording the methods executed by a web page, including property getters and setters, when legitimate content is displayed. Such profiles can then be used to constrain JavaScript behavior, thereby preventing XSS attacks. The system consists of two phases: the training phase during which web page profiles are created and the attack detection phase. Our system is based on the principle that any attack will generate an abnormal sequence of JavaScript method invocations, thereby violating the previously built profile of a web page.

1.3 Contributions

JaSPIn shows that monitoring JavaScript execution is effective at detecting XSS attacks and practical, in that it can be performed online in real-time with a minimum overhead. There are several advantages of our JavaScript based anomaly detection defense as explained below.

No changes to existing applications: More than 70 percent of the 172 million sites on the web today [Net08] use JavaScript. It is practically impossible for any solution requiring changes to every web page in every web site to be well adopted. Thus, our proposal requires absolutely no change to web pages, and no configuration changes on the web server.

Automated generation of stable profiles: Web site specific profiles are built based on the user's browsing patterns. Thus, JaSPIn does not depend on either the end user, or the website programmer to generate profiles or policies manually.

Immunity to Filter Evasion Techniques: Various filters can be evaded by encoding the input into something that the browser understands and is completely valid for the filter. JaSPIn checks for profile violations in the browser at the level of the JavaScript interpreter just before the script executes. Thus filter evading attacks with complex signatures do not fool our XSS detector.

Attack Coverage: The main advantage of our anomaly based approach is that

it does not require prior knowledge of the attacks and can thus detect new intrusions. A cross-site attack actually happens only when the attack code executes. Hence, our anomaly detection engine cannot miss verifying any method invocation. It can even detect advanced attacks vectors not based on input validations such as DNS pinning [Fog07] and the more recent MHTML XSS vulnerability [Fog07]. Although, it is possible for the attacker to mimic normal behavior and craft function calls to execute in the same order as in the profile of a website, such a targeted attack requires way more effort and customization of the attack code.

No whitelists: Generating and maintaining whitelists are cumbersome. Also, approaches which use a whitelist of safe sites to reduce the number of false alarms are opening the XSS attack doors on such sites. In today's linked web, a number of sites purposely transfer information between one another, such as for analytics or advertisements. In such cases, a whitelist based solution would add all of these sites to the list, thereby making vulnerabilities in them exploitable.

As a first step, we have modified the Mozilla Firefox browser to infer JavaScript policies and create profiles of every website visited by a given user. Our anomaly detection engine, JaSPIn has been successfully implemented in Mozilla Firefox. We have found that once a profile has been created for a given web page and learning complete, stability is achieved. Updates to the profile are required only when the web site's JavaScript calls are updated, or the user changes his browsing patterns, such that they do not confer to his previous behavior. Also, our experiments have shown that our technique is able to detect most XSS attacks, while keeping its false positive rate relatively low.

1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 provides background information on JavaScript and the security mechanisms currently in browsers. It also explains what cross-site scripting means, and provides some examples of relevant attacks. We also present two different types of attacks, which although different in nature are similar in effect. It also introduces intrusion detection and reviews related work, finding a place for our system both in the space of web application security and

anomaly detection. Chapter 3 discusses our method of analysing JavaScript method invocations used in our system, JaSPIn.

Chapter 4 discusses the implementation of JaSPIn, which includes modifications to the browser and the development of a plug-in to monitor the policies. Chapter 5 presents results showing how JaSPIn performs. We specifically focus on how good the web profiles developed by JaSpin are in keeping the false positive rate low. This chapter also analyses diverse types of websites and their effect on the functioning of JaSPIn. We then present JaSPIn's attack detection capabilities against some typical real world cross site scripting attacks. The next chapter discusses our findings and comments on the efficiency of JaSPIn, summarizing the contributions of this work and suggesting areas of further work to address current shortcomings. We conclude with a summary of this thesis in Chapter 7.

Chapter 2

Background and Related Work

This chapter provides background information on JavaScript and cross-site attacks. The first section of this chapter gives a brief introduction to JavaScript outlining some of the features of the language that indirectly aid cross-site attacks. The next section takes a look at how web applications have evolved through the years, and describes in parallel the history of cross-site attacks. We then describe the different classes of such attacks and look at them in detail. False positives, false negatives, attack coverage, ease of implementation and ease of use are important considerations when selecting a defense strategy. This chapter looks at various approaches suggested to defend against cross-site attacks. We then look at intrusion detection approaches and suggest anomaly detection as a solution to address some of the limitations of previous approaches.

2.1 Understanding JavaScript

JavaScript is a high level object based, cross-platform dynamically types language with C-like syntax and Scheme like semantics. The most common use of JavaScript is to add functionality to web pages as rendered within a browser; however JavaScript is also used in web servers, flash applets and desktop applications. JavaScript was standardized in the ECMAScript Language Specification, Third Edition [ECM99].

2.1.1 JavaScript Language Features

The core JavaScript language and its built-in data types are the subject of international standards, and compatibility across implementations is very good. JavaScript provides a complete range of basic programming statements, such as assignment statements, if statements, loop statements, and others. JavaScript is a prototype-based language [War01] in which classes are not present, and behavior reuse (known as

inheritance in class-based languages) is performed via a process of cloning existing objects.

JavaScript Functions

Functions are one of the fundamental building blocks in JavaScript.

A JavaScript function definition consists of the function keyword, followed by

- the name of the function
- a list of parameters to the function, enclosed in parentheses, and separated by commas
- the JavaScript statements that define the function, enclosed in curly braces,

For example, a simple function named printhello is shown below:

```
function printhello(string) {
   document.write(" hello "+ string)
}
```

This function takes a string as its argument and displays a hello message.

Defining a function does not execute it, calling it explicitly does. For example, we could call the printhello function as follows:

printhello("Preeti")

JavaScript Objects and Methods

JavaScript objects consists of properties and methods that enable operations on it. Properties are variables that are only accessible via their object, and methods are functions that are only accessible via their object. JavaScript requires all access to properties and methods to go through the objects that contain them. Server-side and client-side JavaScript have pre-defined objects that are specific to their runtime environment. For example, alert() is a method of the window object.

A list of the most commonly used JavaScript methods can be found in Appendix A.

2.1.2 JavaScript in web pages

A web browser provides an ECMAScript host environment for client-side computation including, for instance, objects that represent windows, menus, pop-ups, dialog boxes, text areas, anchors, frames, history, cookies, and input/output. Further, the host environment provides a means to attach scripting code to events such as change of focus, page and image loading, unloading, error and abort, selection, form submission, and mouse actions. Scripting code appears within the HTML and the displayed page is a combination of user interface elements and fixed and computed text and images. The scripting code is reactive to user interaction and there is no need for a main program.

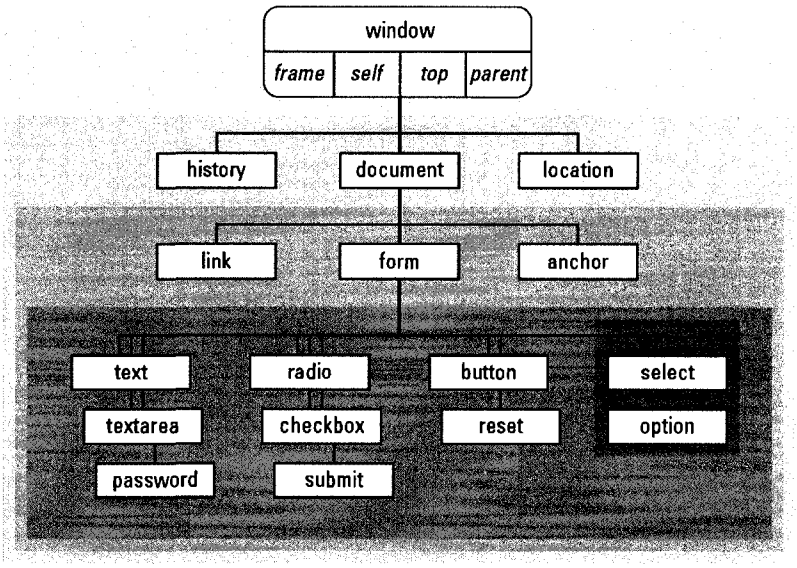

Figure 2.1: JavaScript object hierarchy

Client-side JavaScript is, at its lowest level, several core objects that are created when a page is loaded in the browser. In addition to these core objects, there are also derived objects that are created when certain tags are included on a page. These derived objects inherit some of the various characteristics of their parent object and also allow scripting access to the HTML tags properties. Figure 2.1 gives a graphical representation of the basic client-side JavaScript hierarchy. As shown, all client-side objects are derived from either the Window or navigator objects. All objects on a given page are constructed within the browsers window, hence all objects that

JavaScript can create are descendants of the Window object. JavaScript also has the ability to control HTML objects such as forms, layers, div and frames using the Document Object Model (DOM) [W3C04]; it even has access to parts of the browser not related to HTML, for example, it can determine what version the browser is or which platform the browser is running on. Thus, in JavaScript, methods form the building blocks of accessing both predefined objects and user created objects.

Embedding JavaScript

JavaScript can be embedded in an HTML document in different ways as explained in the sections below:

Using the SCRIPT tag The tags used to begin and end a script are the `<SCRIPT>` and `</SCRIPT>` tags. These can be placed anywhere between the `<HTML>` and `</HTML>` tags in a web page.The opening tag also includes an optional language attribute:

```
<SCRIPT language="JavaScript">
```

The language="JavaScript" command is there so the browser can know that the code that follows is in JavaScript and not another scripting language, such as VB-Script. When absent, the browser interprets the code that follows the `<SCRIPT>` tag as code in the latest version of JavaScript handled by that browser. Javascript code follows this tag, and the `</SCRIPT>` tag is added at the end of the code segment:

```
<SCRIPT language="JavaScript">

.....JavaScript Code.....

</SCRIPT>
```

Using an external source The SRC attribute of the `<SCRIPT>` tag is used to specify a file as the JavaScript source (rather than embedding the JavaScript in the HTML as in the section above). JavaScript files have a .js extension. For example:

```
<HEAD><TITLE>My Page</TITLE>
<SCRIPT SRC="myfunctions.js">...</SCRIPT>
</HEAD><BODY>...
```

This method of including JavaScript in a page is useful for sharing functions among many different pages.

The SRC attribute can specify any URL, relative or absolute. For example:

```
<SCRIPT SRC="http://somewebsiste.com/functions.js">
```

External JavaScript files cannot contain any HTML tags, only JavaScript statements and function definitions.

As event handlers Clicking a button , changing a text field or moving the mouse over a hyperlink are examples of events. Scripts can be made to react to events by defining event handlers such as onChange and onClick.

To create an event handler for an HTML tag, an event handler attribute is added to the tag and JavaScript code is added as the value of that attribute.

```
<TAG eventHandler="JavaScript Code">
```

where TAG is an HTML tag and eventHandler is the name of the event handler.

For example, suppose you have created a JavaScript function called compute. You can cause Navigator to perform this function when the user clicks a button by assigning the function call to the button's onClick event handler:

The following example modifies text to bold when the mouse is moved on top of it.

```
<P ID="boldpara1"
onmouseover="document.all.boldpara1.style.fontWeight= 'bold'" >
This text will turn bold when the mouse cursor is placed on it.
</P>
```

Such multiple entry points for JavaScript make it difficult for the programmer to attack-proof the web sites using input filtering techniques. On the other hand, an attacker needs to find only that one vulnerable entry point to launch a successful cross-site attack. The recent waves of phishing attacks which use cross-site scripting vulnerabilities clearly show that there are many attackers on the Internet looking for easy targets with a vulnerable entry point.

2.1.3 JavaScript Security Model

JavaScript's security model is based upon Java. This section describes the two key security models available in JavaScript: Sandboxing and the Same origin policy.

Sandboxing

Sandboxing [CER06] is a generic security term that applies to any environment that reduces the privileges under which an application runs.

JavaScript programs downloaded into a client computer could have the same access to the system as a local software program. Access of this type would clearly be unacceptable. To safeguard the client from unauthorized access, a number of safeguards are built into the browser to prevent JavaScript running in the browser from gaining access to local machines. This mechanism is collectively referred to as the sandbox, a restricted area in which JavaScript can execute. Client side JavaScript runs in its own sandbox, and does not have access to anything outside the sandbox by design. The language provides no read or write access to local files beyond the highly regulated cookie file. To prevent browser-specific privacy invasions, it is not possible for a script in one window to monitor the users activity in another window, including the URL of the other window, if the page did not come from the same server as the first window. Scripts cannot extend their reach outside of the sandbox to access local file systems and many sensitive system preferences. The script runs only while its containing page is still loaded in the browser. When the page goes away, so does any JavaScript that is part of that page, without being saved to the local disk cache. Sandboxing can prevent scripts from accessing private information outside the browser, but they provide no means for protecting information within the

browser.

Same origin policy

The same origin policy [Rud01] prevents documents or scripts loaded from one origin from getting or setting properties of a document from a different origin. Two pages have the same origin if the protocol, port (if given), and host are the same for both pages.

An origin is not the complete URL of a document. Consider the two popular URLs for Mozillas web sites:

`http://www.mozilla.org` `http://developer.mozilla.org`

The protocol for both sites is http:. Both sites also share the same domain name: mozilla.org. But the browser sees them as sites running on two different servers: www and developer.

Documents from the same server at mozilla.org using the same protocol, have the same origin. A script in a document from one of the two servers would display an access disallowed or permission denied error message if it tried to get the location property of the other document.

Similarly, if a bank's secure pages that use the https: protocol try to access the location object properties of http: pages hosted by the same bank, access is disallowed, even though both these set of pages share the same server and domain name.

The same origin policy applies to external scripts as well. For example, if mywebsite.com includes a script from google.com, the script will execute in the context of mywebsite.com. Thus, the script will have unrestricted access to the page its included from and will be able to issue requests against mywebsite.com, but it will be unable to manipulate google.com documents in other windows and frames. These external scripts are really quite common, and the example google.com relationship is how Google Analytics is used on many sites. The risk associated with external scripts is that they implicitly grant unrestricted page access to a script from a remote site. Hence, compromising the script at google.com would allow attacks against users of mywebsite.com.

Today pages load a lot of code from external sources (ads, videos, counters, music,

etc.) Thus, by including an ad, the ad server has access to anything on the origin's domain.

2.2 JavaScript based attacks

Many security holes have been found with the JavaScript language and its execution environment. Such vulnerabilities range from relatively harmless oversights to serious vulnerabilities that permit access to local files, cookies, or network capabilities.

Also, many JavaScript attacks are possible without violating any security policies, simply by using the language in an undesirable manner. As a simple example, JavaScript is often used to open a customized new window on the client. This feature has been heavily exploited to generate unwanted annoying pop-ups. More seriously, this feature has been exploited in recent times to launch phishing attacks, where key information about the origin of the web page is hidden from users. JavaScript can also be used to cause a denial of service attack on the host either intentionally or by mistake by having an unstoppable infinite loop.

The most prevalent JavaScript based attacks are Cross-site scripting and Cross-site Request Forgery. The principle behind both these attacks is that the hacker gains the ability to insert some arbitrary content into a web page. This content can be used to do things that the trusted site did not intend, like stealing the user's cookies. The difference between CSRF and XSS is the way in which the attack is delivered. XSS relies on the injection of arbitrary data through non-validated input, such as fields from a POST form submission. On the other hand, CSRF depends on browser features to retrieve and execute the attack bundle. These attacks are discussed in detail in the next section.

2.3 Cross-site attacks

2.3.1 Cross-site scripting attacks

Cross-site scripting attack method was first discussed in a CERT advisory back in 2000 [CER00]. But, even today cross-site scripting (XSS) is one of the most common vulnerabilities in web applications. It happens as a result of insufficient filtration of

data received from a malicious person and then sent to third parties. Systems that receive data from users and display it on other users' browsers are very vulnerable to an XSS attack. Wikis, forums, chats, web mail - are all good examples of applications most susceptible to XSS.

Cross-site scripting (XSS) can be defined as a security exploit in which an attacker inserts malicious code into a page returned by a web server trusted by a user. This code may reside on the web server or be explicitly inserted when the user browses to a site, it may contain JavaScript or just HTML, and it may use third party sites as sources or rely only upon the resources of the targeted server. XSS attacks typically involve JavaScript code from a malicious web server executing on a user's web browser.

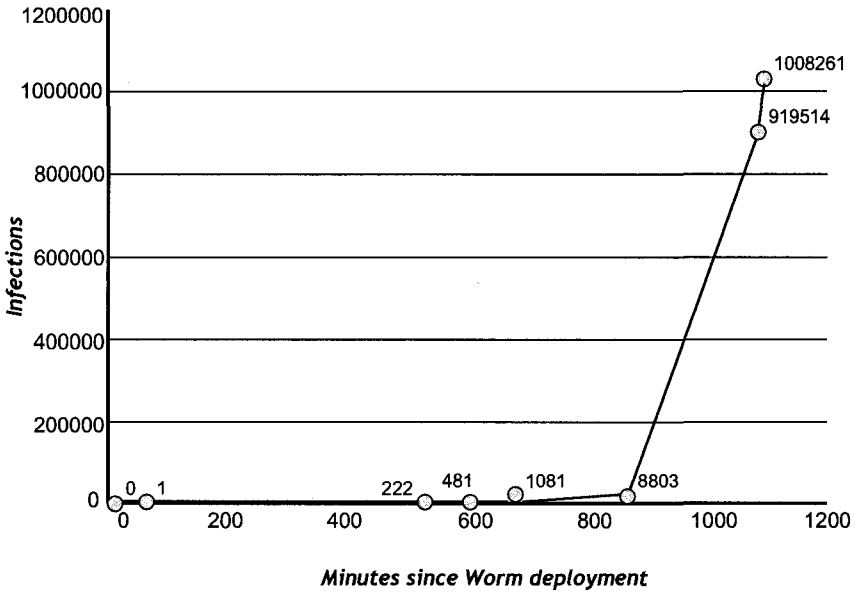

Figure 2.2: Propagation of Samy by exploiting an XSS flaw

The self propagating worm 'Samy' [Som02] is proof of how easy and feasible it has become to exploit XSS vulnerabilities. In October 2005, the highly popular social-networking site MySpace was hit with this cross-site scripting worm that spread

exponentially. The author of the worm created Ajax code on his MySpace site that ran automatically when anyone looked at his profile. Because Ajax can interact with pages users never see, his code pressed all the relevant buttons to add Samy to the victim's friends, and added the words "but most of all, Samy is my hero" to their page. The code also pasted itself into the victim's profile, so that any MySpace user viewing the victim's page would have their page infected. Samy spread extremely fast, as seen in Figure 2.2. Within 24 hours the author of the worm, Samy, had a million MySpace users 'wanting' to be his friend and to whom he a hero. MySpace was forced to shut down to fix this vulnerability, and though the payload of this worm was not harmful, it very well proved how powerful and fast cross site scripting attacks can be.

2.3.2 Dangers of XSS vulnerabilities

The XSS vulnerability can be exploited to do any or all of the following:

- Stealing a user's cookie
- Modifying a web page
- Collecting statistics
- Exploit a browser vulnerability
- Capturing Clipboard Contents
- Stealing History and Search Queries
- Port Scanning and other advanced attacks

Site redirection, access of local files, performing tasks with system privileges etc are some of the other dangers of XSS. In the sections below, a detailed explanation of each of the above is given with an example.

Stealing a user's cookie Cookies are pieces of information generated by a Web server and stored in the user's computer by the server so as to remember the user and her preferences. Many a time cookies contain confidential information including

passwords and banking information. Cookies are also widely used to store session IDs. To obtain the rights of the owner of the ID, the attacker would insert the ID into his or her cookie. In other words, if authentication in a system is based on cookie parameters, an authenticated userss cookies will give the attacker complete access rights. Cookies are a simple way to make the experience of browsing through a site more usable and enjoyable.

Modifying a web page In the ideal secure world, only the domain that created the cookie can access it. JavaScript's same origin policy prevents a data and information loaded from one site of origin from altering the properties of a document loaded from another site of origin. Two web pages are considered to have the same origin if they have the same domain name, port, and protocol. However, in a XSS attack, the attacker's script can access data in the context of the document that is attacked. Many examples of the use of XSS to steal cookies can be found in the literature. We explain a simple real exploit [Rud01] documented in April 2006 here: The very popular eBay web site contained a cross-site scripting vulnerability. When an eBay user posts an auction, eBay allowed SCRIPT tags to be included in the auction description. This created a cross-site scripting vulnerability in the eBay website. Attackers used this vulnerability to redirect auction viewers to phishing sites and to modify the eBay auction page to steal credentials leading to disclosure of passwords, credit card numbers, or other personal information. The following is one course of action for stealing cookies by exploiting eBay's XSS vulnerability:

1. Attacker includes a redirect script in the auction page.

```
<SCRIPT>
new Image().src = "http://myevilsite/?data="+  encodeURI(document.cookie);
</SCRIPT>
```

2. Code gets inserted by server on the victim's browser which executes the same.

3. Code steals victim's session cookie, and the attacker is now logged in to the user's eBay account.

Although simply disabling cookies seems like the easiest solution to prevent cookies from being stored or stolen, many websites rely on cookies to function correctly. Modifying a web page A website which allows users to enter content which is displayed back is susceptible to defacing XSS attacks. An XSS hole in the input page can be exploited by simply embedding the script to deface.

```
<SCRIPT src="(your script location)"></SCRIPT>
- OR -
<SCRIPT>
document.body.innerHTML="<h1>Changing the page style</h1>other code here";
</SCRIPT>
```

By changing the html behind a web page, it can be used as a phishing web site. An XSS vulnerability at the Web site of JPMorganChase.com was found in June 2006 [Kre].

Since the name in the browser address bar does show that the visitor is on Chase's site, it becomes difficult to detect. The content on this page could have been anything the attacker wanted: login form, bank account number, or a link that that redirects the user to any other Web site.

Collecting Statistics A good example to explain how cross-site scripting could allow an attacker to collect statistics is that of a cross-site image embed. The attacker can insert the URL for his image in the target sites code, such that when the image is requested from that server, it will execute malicious code, saving some statistical data before sending the image. JavaScript offers programmers methods for accessing many browser parameters, such as history, referrer etc. This is valuable information to an attacker. Also, the time of visit and IP address of the visitor can prove useful to an attacker.

In addition to frivolous mischief, this kind of attack could be used for serious criminal purposes. A professionally crafted defacement, delivered to the right recipients in a convincing manner, could be picked up by the news media and have real-world effects on people's behavior, stock prices, and so on, to the financial gain of the attacker

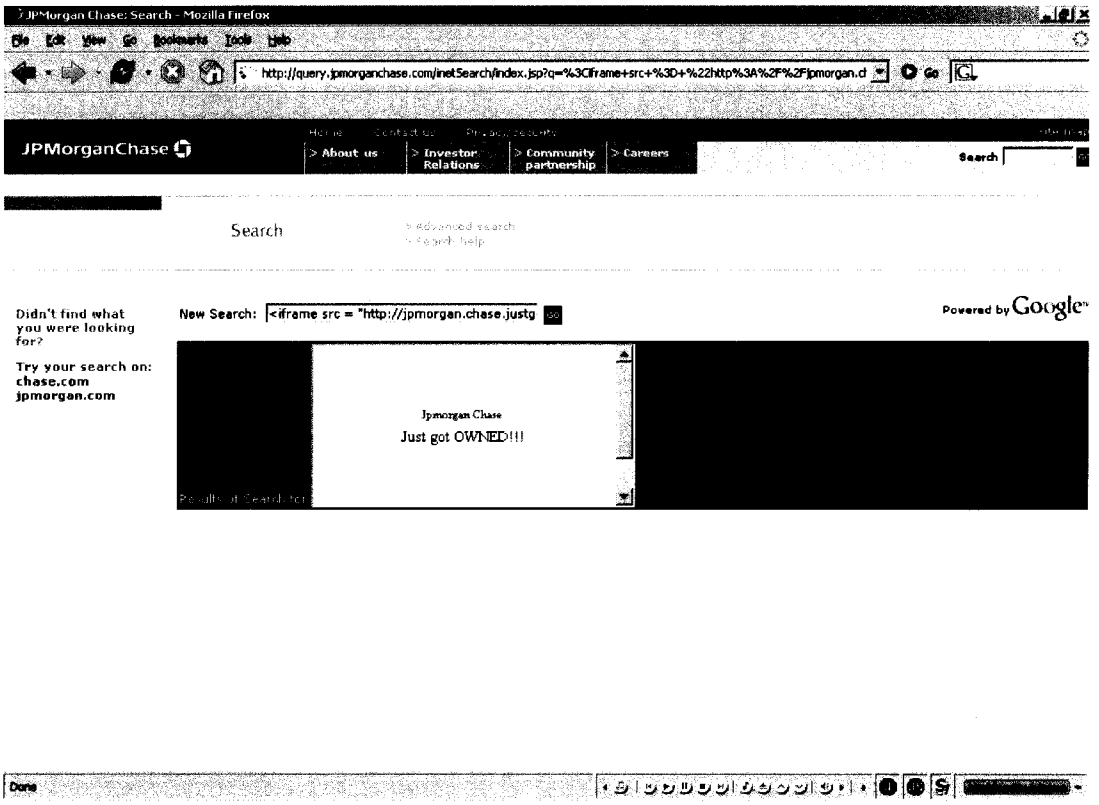

Figure 2.3: JPMorgan Chase.com XSS flaw exposed

Exploiting Browser Vulnerabilities An attacker may be able to exploit bugs in the user's browser or any installed plug-ins via malicious JavaScript or HTML. Bugs within plug-ins such as the Java VM have enabled attackers to perform two-way binary communication with non-HTTP services on the local computer, enabling the attacker to exploit vulnerabilities that exist within other services identified via port scanning. Many XSS vulnerable ActiveX controls have been reported. Adobe Reader and Acrobat versions 7.0.8 and earlier could allow remote attackers to inject arbitrary JavaScript into a browser session [Ado07]

A generic signature for the attack would be:

`http://path/to/pdf/file.pdf#whatever_name_you_want=javascript:your_code_here`

In this case, the attacker does not need to have write access to the specified PDF document. For the XSS attack to work, the only requirement was that a PDF file be hosted on the target site. The rest is just a matter of what the attacker wishes to do

in her JavaScript code.

Capturing Clipboard Contents JavaScript can be used to capture the contents of the clipboard. The following proof-of-concept script will display an alert containing the current contents of the clipboard:

```
<script>
alert(window.clipboardData.getData('Text'));
</script>
```

Monitoring the clipboard periodically while a user works on other tasks might result in all kinds of information being captured. Many users copy and paste their passwords as well.

Stealing History and Search Queries JavaScript can be used to perform a brute-force exercise to discover third-party sites recently visited by the user, and queries that they have performed on popular search engines. This can be done by dynamically creating hyperlinks for common web sites, and for common search queries, and using the getComputedStyle API to test whether the link is colorized as visited or not visited. A huge list of possible targets can be quickly checked with minimal impact on the user.

Port Scanning and other advanced attacks JavaScript can be used to perform a port scan of hosts on the user's local network, to identify services that may be exploitable. If a user is behind a corporate or home firewall, an attacker will be able to reach services that cannot be accessed from the public Internet. If the attacker scans the client computer's loopback interface, he may be able to bypass any personal firewall installed by the user.

Once an attacker is able to identify other hosts after a post scan, a malicious script can attempt to fingerprint each discovered service and then attack it in various ways.

2.3.3 XSS Types

XSS attacks have been classified into two types based on the attack vector: reflected attacks and stored attacks. [CER00] Reflected XSS attacks are more common than the stored type.

Reflected Attack Vector A reflected attack, also known as a non-persistent XSS attack, takes place when malicious code or scripts are output by a vulnerable web server as part of a valid HTTP request. Some common examples of responses are error messages for files not found, search engine results, or submitted web forms. An example of a reflected XSS attack is a case in which an unsuspecting user is enticed to follow a malicious link to a vulnerable server that injects (reflects) the malicious code back to the user's browser. The browser then executes the code or script because the vulnerable server is usually a known or trusted site. Standard methods of delivery for XSS exploits are via e-mail, instant messenger applications, or search engines.

Stored Attack Vector A stored attack, also known as a persistent attack, takes place when the malicious code or script is permanently stored on a vulnerable or malicious server using a database, blogs entries, newsgroup or web forum posts, or any other permanent storage method. An example of a stored XSS attack is a case in which a user requests the stored information from the vulnerable or malicious server, which then injects the requested malicious script into the user's browser. The browser then executes the code or script because the vulnerable server is usually a known or trusted site. For example, an attacker can post a message containing the malicious script to the message board, which stores and subsequently displays it to other users, causing the intended damage.

Our solution is able to detect both reflected and stored attacks as shown in Chapter 5.

2.3.4 Cross Site Request Forgery

XSS exploits the users's trust in a Web site, while CSRF [Law07] exploits the trust a Web site has in its users. Many a time the user is unaware of what is happening in

the background, and does not realize that the attacker has unauthorized access to his system and is sending requests to the user's web page provider by pretending to be the user. To authenticate and thus gain access to a Web site or corporate intranet, a hacker uses either the compromised computers IP address or cookies that the site placed on the machine.

The sequence of steps involved in a basic cross-site request forgery are as follows:

1. The victim is logged in to the target site
2. The attacker posts a link to a malicious site on the targeted site (or link is provided through some other means)
3. The victim browses to the malicious site
4. The malicious website submits a form that modifies sensitive data with the action pointing to the target site
5. Form submission is accepted if the victim is still logged in to the target site. The attack is successful.

The attacker can also modify the user's credentials when logged in, thus gaining access to all sensitive content. CSRF attacks are not necessarily limited to submitting a single fraudulent request. Multiple forms can be automatically submitted sequentially via JavaScript as long as no new identification parameter is required and the attacker is aware of the sequence of these forms.

One well-known example of an CSRF flaw was found in the eBay application by Dave Armstrong in 2004 [Pro03]. It was possible to craft a URL that caused the requesting user to make an arbitrary bid on an auction item. A third-party web site could cause visitors to request this URL, so that any eBay user who visited the web site would place a bid. Further, with a little work, it was possible to exploit the vulnerability in a stored OSRF attack within the eBay application itself. The application allowed users to place ¡img¿ tags within auction descriptions. To defend against attacks, the application validated that the target of the tag returned an actual image file. However, it was possible to place a link to an off-site server that returned a

legitimate image at the time the auction item was created, and subsequently replace this image with an HTTP redirect back to the crafted XSRF URL. Thus, anyone who viewed the auction item would unwittingly place a bid on it.

2.3.5 Evolution of cross-site attacks

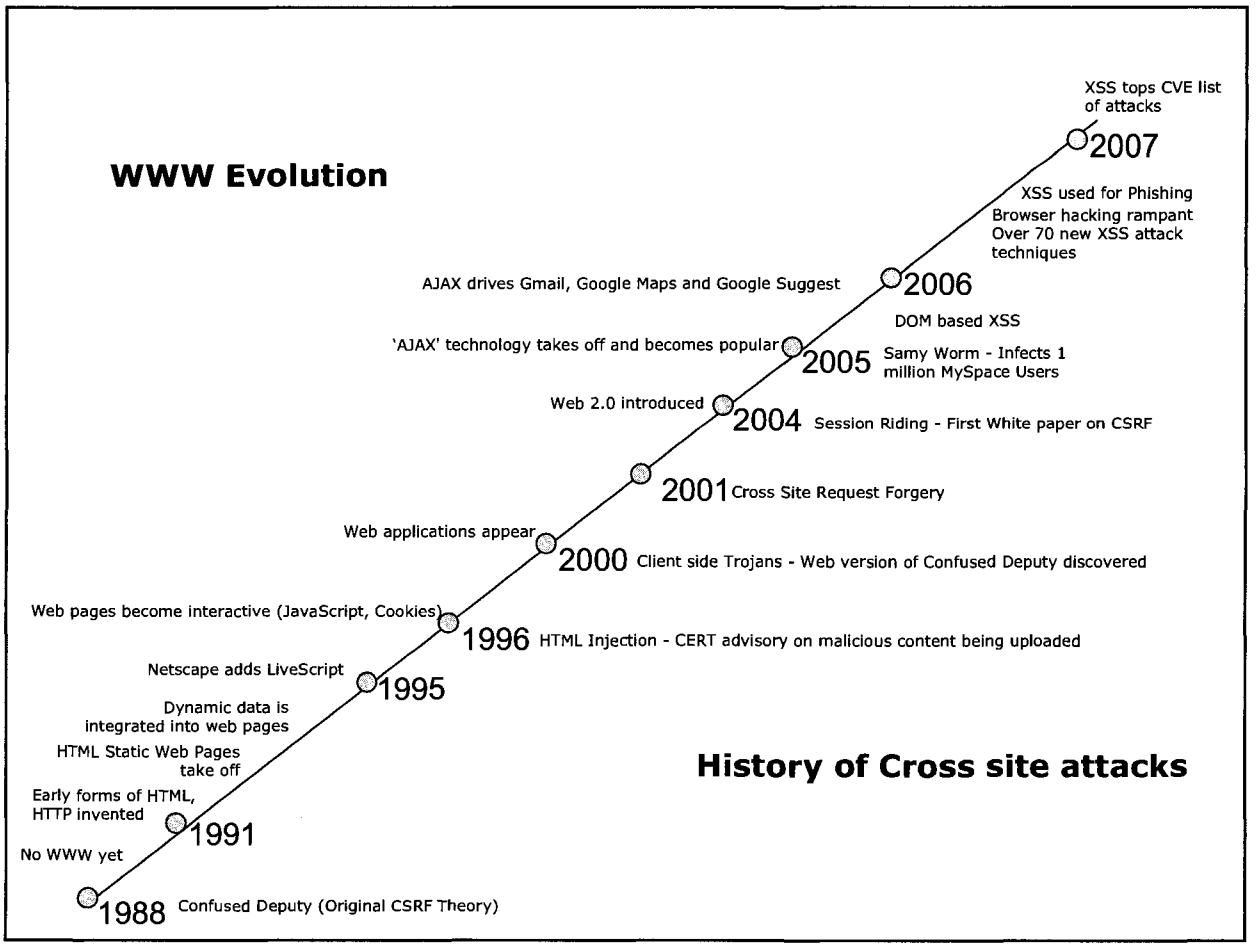

Figure 2.4: History of Cross-site attacks alongside the evolution of the web

Figure 2.4 presents a birds eye view of the evolution of the World Wide Web from a medium to facilitate sharing information among researchers to its current interactive and dynamic state. Alongside, it also outlines the history of cross-site vulnerabilities which is as old as the web itself.

In fact, even before the World Wide Web came into existence, Norm Hardy published a document in 1988 explaining an application level request forgery issue called

confused deputy [Har88] where a source of authority can be confused into permitting something that should not happen to happen. Malicious JavaScript today is able to confuse web applications to do the exact same thing. This correlation and the roots of Cross-site request forgery attacks are explained in a post to bugtraq in 2000 that explains how ZOPE, an open source web application server was affected by a confused-deputy web problem that we would define today as a Cross-site Request Forgery vulnerability.

Static web pages constituted much of the web in the early days of the internet. These were essentially repositories of information containing static documents, and web browsers were invented to retrieve and display those documents, as shown in Figure 2.5. The flow of interesting information was one-way, from server to browser. Any security threats arising from hosting a web site related largely to vulnerabilities in web server software, and cross site attacks on the web were unknown.

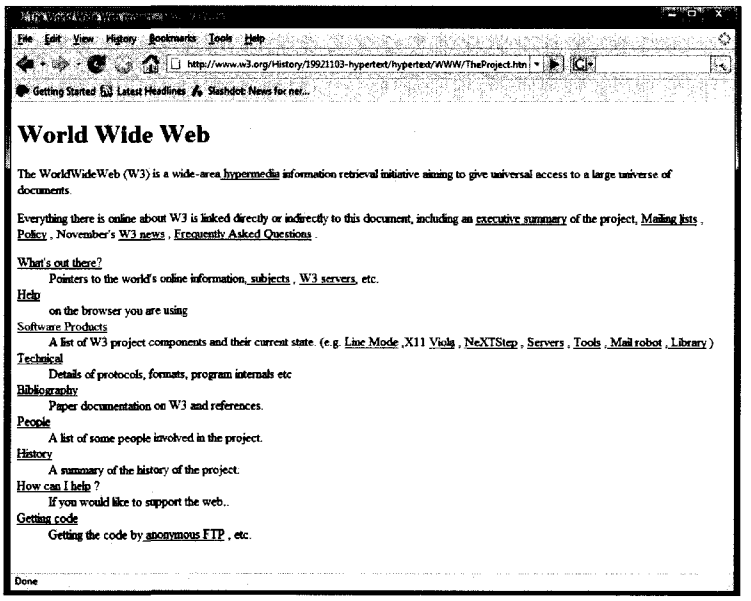

Figure 2.5: A static 1992 web page, which is an updated version of the first web page on the internet

LiveScript, an earlier version of JavaScript was added by Netscape in 1995, but did not become very popular as only Netscape products supported it. Once JavaScript was introduced, programmers were able to create interactive web pages with special effects, and hackers discovered an unexplored world of vulnerabilities. The very

first known cross site attacks used HTML frames within the same browser window. JavaScript was used to read from one frame to the other, thereby enabling stealing of cookies and other confidential information. To prevent further such attacks, Netscape introduced the "'same-origin policy"', a policy preventing such leak of information among different websites. The same origin policy is explained in detail in Section 2.1.3. By the year 2000, many cross site vulnerabilities were discovered, and Microsoft acknowledged that although this was not an entirely new issue the overall scope of the issue was larger than previously understood. It was in 2000, that the term Cross-site scripting (XSS) was coined.

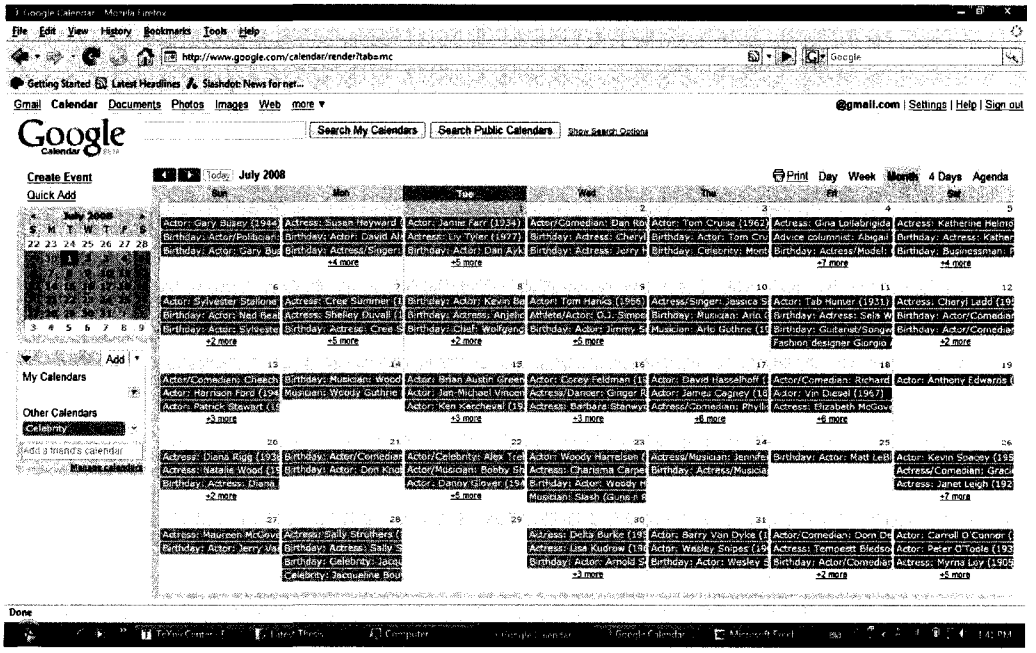

Figure 2.6: The Google Calendar is a good example of a popular web application used today

Many developers and security experts did not recognize the power and potential of JavaScript till 2005. In 2005, AJAX took off and a number of web sites started using AJAX techniques to improve their user experience. Applications such as Google Calendar 2.5 use JavaScript extensively to enhance usability by avoiding page refreshes.

The rise of JavaScript usage led to the rise of cross-site attacks as well. In October of 2005, a popular social networking website MySpace had to be shut down when a cross-site scripting vulnerability in the site was exploited. The worm was spreading

at a rate of 1,000 users every few seconds before MySpace shut down its site. A few months later, in 2006, we saw the birth of JavaScript key loggers, port scanners, trojan horses, intranet hacks etc. Cross site scripting vulnerabilities were being discovered in many websites. To date, roughly 25561 XSS vulnerabilities have been found and reported [FP07].

2.4 Current browser security mechanisms and their limitations

This section looks at some of the security mechanisms available in browsers today. They are anything but complete, and attackers have found many ways to circumvent the same.

2.4.1 Zone security

Browsers today claim to have the ability to set JavaScript policies for any particular website. In Internet Explorer, for example, both versions 7 and 6 offer very minimal fine-grained security against JavaScript based attacks. Zone level security can be configured in the browser options, as shown in Figure 2.7, and allows for the user explicitly adding a website to a particular zone; thereby enabling or disabling all JavaScript on that site.

This approach relies on the user to make all the decisions; and provides no help. Also, since most sites today require JavaScript to be enabled to work properly, disabling all JavaScript may not even be an option for the user.

2.4.2 XSS Filter

XSS Filter is an IE8 component with visibility into all requests / responses flowing through the browser. When the filter discovers likely XSS in a cross-site request, it identifies and neuters the attack if it is replayed in the servers response. Users are not presented with questions they are unable to answer IE simply blocks the malicious script from executing.

The XSS filter claims to detect only reflected XSS attacks. Since no user studies are available as of this writing on the XSS filter, we are unable to determine its

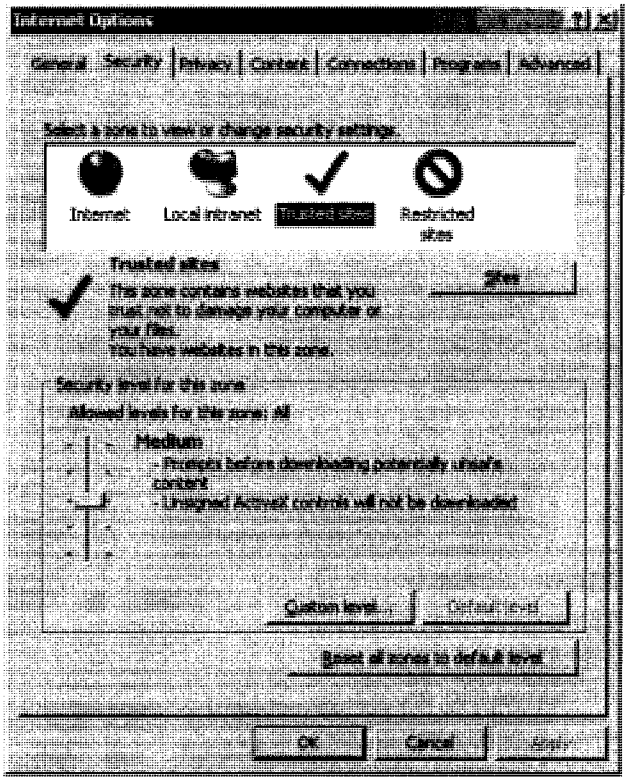

Figure 2.7: Current solution to set security policies for websites in Internet Explorer 6 and 7.

efficiency. Also reflected attacks are only one of the types of XSS attacks present today.

2.4.3 Disabling JavScript

For end users, the most effective way to prevent cross-site scripting attacks is to disable all scripting languages in their web browsers. The downside of this is the resulting loss of functionality. Many web sites today heavily utilize JavaScript for functionality, and may not work properly if JavaScript is disabled. Users can also be selective about the websites they visit.

Many online banking portals such as the one shown in Figure 2.8 are fully dependent on JavaScript to function as required. Figure 2.8 shows the online banking portal of TD Canada Trust accessed through a browser with JavaScript disabled. The page fails to function as required and a blank page is all that displays.

As turning off JavaScript completely breaks the functionality of many modern

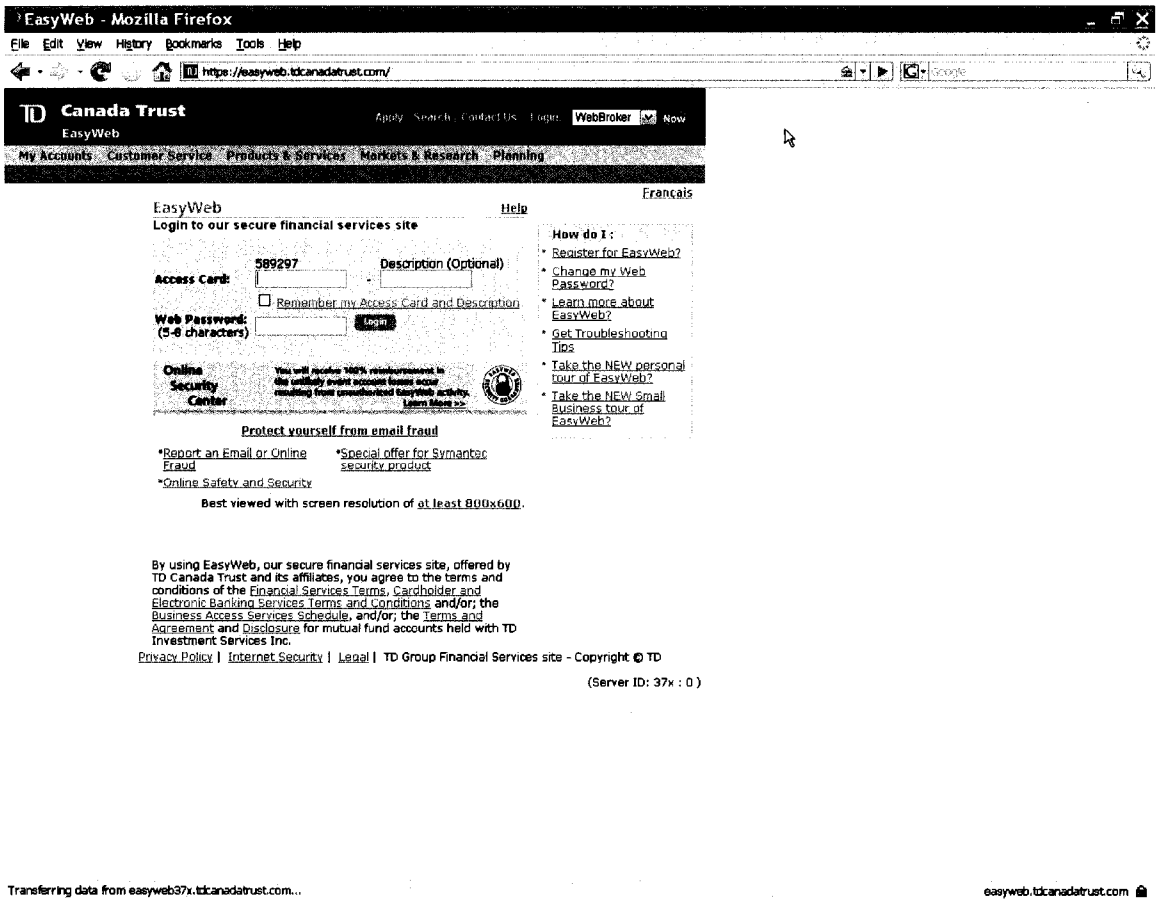

Figure 2.8: Online banking website with JavaScript enabled.

websites, the usage of browser-tools that allow per-site control of JavaScript like the NoScript extension [Mao08] may be better.

2.5 XSS attack detection and prevention techniques

Researchers in the past have studied various mechanisms to protect against JavaScript based attacks. They are implemented either at the client or on the server and can be used to either detect or prevent cross-site attacks. The following section explains the different defenses suggested in the past couple of years. We look at techniques that were suggested first, followed by ones that were suggested at a later date. Recent approaches are outlined at the end of the section.

Figure 2.9: Online banking website with JavaScript disabled. No functionality is available as well.

2.5.1 Server side defenses against XSS attacks

Writing safe code

Cross-site scripting vulnerabilities can be reduced with proper filtration on user-supplied data. Some simple techniques include:

- Encoding output

  All non-alphanumeric client-supplied data should be converted to HTML character entities before being redisplayed to a client. For example, the less than character would be converted to <.

  Web page developers are responsible for modifying their pages to eliminate these types of problems.

Technique	Papers	What	Advantages	Disadvantages
Writing safe code		Techniques such as input validations, output encoding etc.	Programmer has control, Highly effective if done properly	Error prone, Tremendous effort required. Evasion is easy
Application level firewall	[SS02a]	Incoming input data sanitized according to rule set	Flexible rules	Rules are in a separate language. Dependence on system admin
Server side proxy (AppShield)	[Inc02]	Inspects inputs to the application and blocks potential threats	No application specific configuration	Cannot detect complex attacks., or zero-day attacks
Instruction-set Randomization	[KKP03]	Creates an execution environment that is unique to the running process.	Applicable to a wide range of code injection attacks	Can be circumvented, Performance Penalty, Cannot detect all XSS attacks
Static Taint Analysis	[HYH+04]	Propagate taint information through PHP program	Successful for PHP code	Requires programmer effort. Can miss code paths
String Analysis	[Min05]	Searched the source code for string patterns	Effective for simple attacks	Easily evaded
Taint-enhanced policy enforcement	[XBS06]	Application based policies combined with taint checks	Flexible, applicable to other web attacks	Similar to input filtering. Dependant on policies. Can be easily evaded.
Static code analysis (Pixy)	[JKK06]	Taint based PHP source code analysis	Web version available, Tested on more than 50 real world exploits	Cannot detect complex attacks. False positive rate of 50%
Http only Cookies (IE 6, SP1)	[Cor06]	New property for Cookies making them inaccessible using code	Easy coding for newer applications	Legitimate uses are also banned, Only in Internet Explorer 6.0 +
Approximating Automatic Data/Code Separation	[JB07]	Uses string masking to persistently mark legitimate code in string values	Limited changes to application's source code	Still in preliminary stages
Combination Static Analysis	[WS08]	Adapted String Analysis using formal methods	Detects many known XSS vulnerabilities, No effort by end user	Manually written input validation functions, Does not detect DOM-based XSS, Cannot handle complex dynamic code, Programmer dependent

Table 2.2: Different Server side XSS defenses suggested over the years

- Adding double quotes around all tag properties

  A common XSS attack technique is to exploit query string variables. For example, scripts can be added to query string variables by closing the $< a >$ tag, and appending with the $< SCRIPT >$. Web developers can defend against this attack by placing optional double quotes around each tag attribute.

  Ex: For the URL

```
<a href=http://www.mysite.com/detail.asp?
id=<%= request.querystring("id") %>>
```

  The query string "'id"' can be used as an XSS entry point by giving it a value such as

This attack can be prevented just be enclosing the href attribute in double quotes.

```
<a href="http://www.mysite.com/detail.asp?
id=<%= Server.HTMLEncode (request.querystring("id")) %>">
```

Even this tremendous effort is not enough to completely erase the threat of a cross-site scripting attack, as evasion is possible. For example, if a filter disallows the $< SCRIPT >$ tag, it can be evaded by using event handlers to directly include JavaScript code without using a $< SCRIPT >$ tag as shown below.

```
<BODY onload="alert('Sample code for XSS')">
```

Further, style sheets can also be used to add JavaScript to a page.

```
<LINK REL="stylesheet" HREF="javascript:alert('Sample code for XSS');">
```

Multiple entry points for JavaScript make it almost impossible for the programmer to write safe code. Also, recent XSS attacks using PDF documents, Flash objects, images etc as input points make it even more difficult to write XSS attack proof code.

Application level firewall and proxies

An affordable start to intrusion detection can be made using the Firewall. Recently, many commercial routers can be configured as an IDS sensor to track and audit the packet flow through the router. When a packet or a number of packets match a certain signature, it will respond to that match in the way you have configured it to respond.

Scott and Sharp developed a very expressive web security policy language that allows one to specify what are legal HTTP and HTML requests [SS02a]. The goal is to protect specific web applications. They also developed this into an anomaly IDS called SPECTRE that infers this policy through observing the behavior of the web application [SS02b]. However, their system requires the correct identification and validation policies for each individual entry point to a Web application. This

is a difficult security task that requires careful configuration by highly technical and experienced individuals. Another problem with this approach is the increase in server response time. Thus, if the number of hits increases, the dynamic generation of web pages will slow down the server substantially.

The Sanctum AppShield Firewall that later became IBM's AppScan is a commercial server side proxy solution that apparently does not need security policies [Inc02]. AppShield executes default filter operations on all user provided data in order to remove potential XSS attacks. Opposed to Scott and Sharps approach, AppShield requires no application specific configuration which makes it easy to install but less powerful. [SS02a] report that AppShield is a plug-and-play application that can only do simple checks and thus can only provide limited protection because of the lack of any security policies. Recent reviews of AppScan have criticized its usability and indicate that it is not able to detect all types of Cross-site attacks.

Code modification

Instruction Set Randomization (ISR) has been proposed by Kc et.al as a defense against code injection attacks in general. It defuses all standard code injection attacks since the attacker does not know the instruction set of the target machine. The key to the success of ISR is the randomization key. Thus a motivated attacker may be able to circumvent ISR by determining this key. Also, ISR is not transparent to developers and requires the transformation of application code. Although proposed in general for code injection attacks, previous work on ISR defenses do not seem to handle JavaScript injection, and are ineffective against XSS attacks.

String analysis methods generate a formal language representation (e.g., a context free grammar) of the possible string values a program may generate at a certain program point. Minamide [Min05] describes an application of string analysis to statically detect cross-site scripting vulnerabilities and to validate pages generated by web applications written in the PHP language. He first extracts a grammar from a PHP program considering assignments as production rules, the grammar is then transformed into a context-free grammar, and, finally, this is used to validate the desired properties. He claims to support almost all string operations available in PHP,

including regular expression-based replacement. He suggests using this analysis to check for XSS vul nerabilities, but his proposed technique checks the whole document for the presence of the $< script >$ tag. This is a rudimentary approach because web applications often include their own scripts, and because many other ways of invoking the JavaScript interpreter exist.

Jovanovic [JKK06] recently presented another server side technique to detect XSS attacks. Pixy is a Java program that performs automatic scans of PHP 4 source code, aimed at the detection of XSS and SQL injection vulnerabilities. Pixy takes a PHP program as input, and creates a report that lists possible vulnerable points in the program, together with additional information for understanding the vulnerability. He has also implemented a web version of the tool which is available to the public. Although efficient in detecting XSS vulnerabilities in the applications tested by the authors, Pixy needs one file as input, where all file inclusions have been resolved and included source code has been embedded. Also, the observed false positive rate reported is at around 50 percent, which puts a lot of emphasis on manual analysis of every detected vulnerability. This is not only time consuming, but also not very helpful to a novice programmer who needs to decide if the detected vulnerability is a real one.

Introduced by Microsoft in Internet Explorer SP1, the Http only attribute marks a cookie inaccessible through script. Although this prevents the attacker's script from accessing cookies, it also prevents legitimate code from doing so. Also, the cookies are still visible using XMLHTTPRequest, and thus can be accessed by using AJAX approaches. The benefits of using this approach is that httpOnly cookies are not visible using simple JavaScript methods such as document.cookie and that makes XSS a bit more difficult when using it in context of credential theft. The disadvantage is that it doesnt work in all browsers and in some old browsers, like IE5.5 on Mac and WebTV, it can actually cause the page to fail to load.

SMask [JB07], developed by Johns and Beyerlein is a novel technique proposed against script injection attacks. The authors observe that scripting attacks are possible due to data being run as code. Thus, by using string masking to mark legitimate code in string values, SMask is able to identify code that was injected during the

processing of an http request. SMask works transparently to the application and is implementable either by integration in the application server or by source-to-source translation using code instrumentation.The approach proposed by the authors to determine if a string contains code is rather coarse. They acknowledge the high false positive rate. Also, since their current XSS tests are limited to three applications, one cannot comment on their detection capabilities either.

Taint based approaches

Huang et al. describe WebSSARI, a white-box tool that uses static code analysis and run- time inspection to locate and partially fix input-based web security vulnerabilities. Their work was among the first such static analysis based work. Their type-based tool considers any data derived from tainted input to be fully tainted. WebSSARI inserts calls to sanitization routines that filter potentially dangerous content from tainted values before they are passed to security-critical functions.

The main advantage of static analysis (as compared to runtime techniques) is that all potential vulnerabilities can be found statically, while its drawback is a relative lack of accuracy. In particular, these techniques typically detect dependencies rather than vulnerabilities.

WebSSARI is reported to have a false positive rate between 26-30 percent. It performs signature-based filtering to sanitize untrustworthy data, hence the effectiveness of the tool is only as good as the input filters and can be evaded. Authors of WebSSARI note that even if all possible attack patterns can be enumerated, real-time filtering would be impractical and have a huge performance impact.

WebSSARI incorporates a compile-time verification algorithm that statically approximates runtime state. Web applications are mostly written in scripting languages such as PHP, ASP, Perl, JavaScript etc. Scripting languages are not compiled into executables but executed directly by interpreters. Predicting the runtime is complex in such applications as the they interact with the underlying interpreter at runtime. The approximation mechanism used in WebSSARI may not be adequate and accurate in current AJAX applications.

Xu, Bhatkar and Sekar [XBS06] have proposed augmenting traditional security

policies with information about the origin of each byte of data used in security-sensitive operations. With this information, their security policies can distinguish between accesses made by an application on its own accord, and accesses made on behalf of untrusted users; thereby detecting a range of attacks including cross-site scripting. They report only the analysis of SquirrelMail for XSS vulnerabilities, and give only a single example in their paper for XSS vulnerability detection. As with any policy based approach where policies are not learnt but specified, their system is only as good as the policies.

Combination static analysis

Su and Wasserman [WS08] combine the work on tainted information flow with string analysis to detect both SQL injection and XSS attacks. They introduced a tool to detect only SQL injection vulnerabilities in their paper. Their analysis checks web applications against the policy that no untrusted data should invoke the JavaScript interpreter. This is represented as a black-list rather than a white-list. Thus, maintaining an accurate black list is key to the success of their method. Omissions in black-list policies will imply loss of detection efficiency.

2.5.2 Client side defenses against XSS attacks

Client side proxy

[IEKY04] Ismail et al. introduced a web-proxy based IDS whose goal was to mitigate XSS vulnerabilities [IEKY04]. It is implemented as a client-side proxy that compares requests and responses and disables them if malicious characters are detected. It was fairly primitive, however, and relied mostly on heuristics and had no learning component. Also, it was able to protect only against reflected cross-site scripting attacks. It does not prevent cross-site request forgery attacks or other complex cross-site attacks.

Noxes, also implemented as a proxy hosted on the client, prevents connections to websites that are not allowed by some usage heuristics as well as the security policy specified by the firewall rules [KKVJ06]. These rules are generated in three ways: manual creation, by prompting, and through training. The training feature most

Technique	Papers	What	Advantages	Disadvantages
Anomaly detection of web based attacks	[KV03]	Log file with HTTP requests analyzed	Based on a learning model, hence requires no changes to the application	Reliance on web access logs, not tested across all XSS attack types
Client side proxy	[IEKY04]	Monitors HTTP requests and responses of the user	Attack information is shared via a central repository	Difficult to adopt, user interference required, transmission interception possible
Monitoring JavaScript code execution	[HV05]	Intrusion detection system built around a FSA	Permits fine-grained policies on JavaScript execution	Many implementation details are unclear. Method to generate policies not explained.
In browser web proxy (Noxes)	[KKVJ06]	Proxy with manual and automatically generated rules	Flexible configuration of rules	Protects mainly against cookie theft, High false positives, may fail with AJAX apps.
Code-rewriting (BrowserShield,CoreScript)	[RDW+07] [YCIS07]	Rewrite scripts according to a security policy prior to executing them in the browser.	Fairly complex policies	Can be easily evaded. Performance penalty, Common policies for all sites.
Dynamic Data Tainting	[VNJ+07]	Tracks the use of sensitive information in the JavaScript engine	Effective for simple attacks	High false positive rate for sites with multiple sources, Heavy user interaction
Browser-Enforced Embedded Policies (BEEP)	[JSH07]	Checks allowable JS functions as specified in the web page's header	Good attack detection	Modifications required to every web page, subject to mimicry attacks
Mutation-event transforms, or METs	[UELX07]	Client side flexible policies	Based on execution monitoring	Still in preliminary stages - no implementation yet
Session Safe	[JB07]	Combination of different solutions for various session stealing XSS classes	Works based on removing an attack requirement, not input sanitation	Attacks against the IDS possible

Table 2.4: Client side XSS defenses

reflects the operation of our system, and Noxes, when used with this feature, can be considered an anomaly intrusion detection system.

Our system is different from Noxes and other similar systems because they act at the phenomenological level of the script text rather than at the level of the JavaScript interpreter. Our system observes attacks directly at the level of their execution.

Code-rewriting (BrowserShield,CoreScript)

BrowserShield [RDW+07], CoreScript [YCIS07] and other similar tools use automatic JavaScript rewriting to enforce security policies and monitor the runtime behavior of JavaScript applications. In BrowserShield, trusted JavaScript functions are inserted to mediate access to the document tree by untrusted scripts. CoreScript policies are specified as a kind of edit automata. Both these tools BrowserShield and CoreScript also require policy specifications, and hence face the same challenges as other policy based approaches where the policies are not inferred. Also, since they

parse HTML and JavaScript in the page, the performance impact is significant. Also, while the authors suggest specifying site-independent policies, it is unclear how this can be achieved, as something valid for one site may not be for another.

Dynamic Data Tainting

Vogt [VNJ+07] proposes detection of malicious flow of sensitive information to a remote attacker using mostly dynamic, language-based taint propagation. This approach addresses only one class of XSS attack; it does not mitigate the damage of other XSS-based attacks, such as port-scanning (where the sensitive information does not appear in the form of data),web page defacement and browser resource consumption. Also, his current implementation does not track data flow using transfer methods like the XMLHttpRequest object, which is extensively used in AJAX applications. Unlike taint propagation, our approach is based

Intrusion detection of based attacks

Kruegel and Vigna's work [KV03] using web server log files to learn the normal behavior of a web page is among the first applications of anomaly detection to defend against web based attacks. The analysis techniques used by the tool take advantage of the particular structure of HTTP queries that contain parameters. The parameters of the queries are compared with established profiles that are specific to the program or active document being referenced. Although only two cross-site scripting attacks were tested, and not much can be said about the XSS vulnerability detection efficiency of their approach, this is a pioneering work in applying anomaly detection to the web. Although out work is very different in terms of the learning methodology, it is also an anomaly based policy inference engine.

In [HV05], Hallaraker and Vigna propose an auditing system for Mozilla's JavaScript interpreter. The authors present in their work the implementation of this approach and evaluate the overhead introduced to the browsers interpreter. The overhead appears to be on the higher side which make this approach unscalable when analyzing non-trivial JavaScript based routines.

Their system however is most similar to ours in terms of its implementation. It's an

IDS built around a JavaScript auditing system. However, the IDS is built on detecting specific attacks conforming to custom-coded FSA. They do not report false positive rates or their effectiveness at non-synthetic attacks. All IDSs can be considered systems for notifying administrators of security policy violations. However, the form the security policy takes differs greatly depending on the type of the IDS. Signature-based IDSs are very lenient. The security policy is a specification of what is not allowed. Specification-based systems encode the security policy in the specification, but require administrators to manually specify that policy. Learning systems like anomaly IDSs infer what is allowed through observation and build a security policy from that.

Anomaly intrusion detection is usually not considered policy inference because the normal profile is not a perfect reflection of security policy. In the system call example, the normal profile does not contain all sequences that are seen during normal runs, creating false positives. Also, because it usually focuses on only one or a few features of execution, attacks that do not alter those features can succeed. However, these flaws are often found in manually specified policies as well. There are well-defined policies for JavaScript in web-browsers, but they are not specific to particular web-applications or web-documents. Also, XSS attacks do not violate these policies in any way. Adding web-application and web-document specific policies is a potential solution to XSS attacks.

Jim et al. recently demonstrated a client-based solution for embedding custom security policies in web pages through minor browser modifications and a special JavaScript object, analogous to Java's SecurityManager class [JSH07]. It allows very expressive security policies, but they demonstrate two: white-listing of specific nodes in the DOM and black-listing of DOM nodes through the use of non-execute tags. This is different from our work in that it works at the DOM level, not the interpreter level, and they do not attempt to infer policies. However, it is similar in that is allows web-document specific policies.

Deployment poses a practical limitation for BEEP, because both the client and the server must use it in order for it to work.

Other recent approaches

Mutation-Event Transforms, or METs, [UELX07] are proposed as a simple, flexible client-side mechanism for security policy enforcement. With METs, the Web server specifies programmatic security policies by including code for JavaScript callback functions in the first script executed in returned Web pages. The Web client enforces these policies by invoking the callbacks on each page modification, or mutation event, including initial page loading.This is quite similar to BEEP [JSH07], and requires both client and application modifications. The proposal is still in its infancy, hence there is no data available on attack detection capabilities. However, the effort required to deploy this approach does appear to be very high.

SessionSafe [JB07] is a very different approach to XSS attack handling and concentrates on three types of XSS attacks. It provides a combination of counter measures for Session hijacking. Thus, not all XSS types are handled, for example, a malicious attacker can still be able to modify the pages appearance or redirect form actions. At present, SessionSafe is still in its early stages and no data is available on false positives or detection capabilities; nor is there any specific plan in place to integrate it into established frameworks and application servers.

Recently, Same Origin Mutual Approval (SOMA), a new policy for controlling information flows that prevents common web vulnerabilities was proposed by Oda et.Al. By requiring site operators to specify approved external domains for sending or receiving information, and by requiring those external domains to also approve interactions, page content from malicious servers is identified and prevented from being executed. Approaches such as SOMA complement our solution and can form a layered defense as explained in the Threat Model section 3.2 in Chapter 3.

2.6 Relevance of Intrusion Detection to cross-site attack detection

To help better understand the motivation for our solution, this section provides an insight into intrusion detection techniques and their application to the web. Our work has been inspired by some of the previous work in the field of anomaly detection. Although anomaly detection techniques have been implemented for OS level

intrusion detection and malicious URL / packet detection, JaSPIn is the first known implementation of a complete anomaly detection based policy inference engine for cross-site attacks.

2.6.1 Overview of Intrusion Detection Systems

An intrusion is defined as a sequence of related actions performed by a malicious adversary that results in the compromise of a target system [KVV04]. In the context of XSS attacks though, the target can be either the web server, or the end user. An intrusion detection system (IDS) is a method of detecting attacks by monitoring network or system events. To find the most suitable intrusion detection system for detecting web based attacks, we need to understand the classes and types of IDSs. While building different types of intrusion detection systems, researchers have used one of three approaches [Axe00] : signature detection, specification, or anomaly detection. Some hybrid solutions have been proposed, where the strengths of one method is exploited to cover the weaknesses of another. Also, based on the data source, IDSs can be classified into two groups: network based and host based. The following sections will give a brief overview of these IDSs, with emphasis on web-based attacks. This will facilitate a better understanding of our decisions to build a browser based policy inference engine to detect web attacks.

2.6.2 Types of intrusion detection systems based on data source

Host and network based IDS differ in their input and location. A host-based intrusion detection system is typically designed to inspect input on a particular machine, while a network-based IDS monitors network activity, and therefore, its takes network traffic as its input. Implementing an efficient network based IDS for the web is difficult due to overwhelming traffic. Also, the IDS can be dodged by obfuscation, partial or complete encryption, fragmenting the source or overloading, such that not all input is validated, all of which are common on the web. Also NIDS were typically designed to work at the TCP/IP level, and are not as effective at the HTTP level.

Host- based IDSs for web applications can be located either on the server or the client. Client-side techniques in the area of web security are primarily used by the

consumers of web applications, rather than by the providers. It is interesting to note here that client-side tools are preferable for security-aware users who wish to achieve a higher level of protection across a wide range of visited web sites. Not every web server is efficient at reacting to security threats immediately. For these cases, an active protection on the client side is able to fill at least a part of the resulting gap. However, the downside to a client-based approach is that the user needs to install additional programs and patches to achieve the desired level of protection.

2.6.3 Types of intrusion detection systems based on detection approach

Signature based systems (also termed misuse-based systems) scan system input for known attack signatures. What this means is that, the attack has been previously studied, and a specific characteristic of the attack has been identified either manually or automatically. This is known as the signature of the attack. The IDS stores a collection of such signatures, and raises an alarm when it finds input data that matches the signature. Most signature based systems make use of some form of pattern matching. Classical virus scanners are signature based. The speed of a signature based system is dependent on the number and complexity of its signatures. Also, a signature-based IDS is only as good as its database of stored signatures: the larger the database, the better its attack detection capability. However, the more advanced the database, the higher the processing time. One way to reduce processing time is to split the feed, but this increases complexity and cost.

Specification based systems compare the behavior of the program to be protected to a set of pre-defined policies. These policies are typically generated by a team of experts who study the program and decide on the allowed actions hat program can take. Sometimes, policies can also describe what is not allowed, instead of what is allowed. Once again, the speed of a system is dependent on the complexity and depth of its policies. Also, the IDS is only as good as the policies that it checks against. This means that the level of expertise required to generate good specifications is quite high, and any slight change in the program would require a review of the specifications. This costs time and money.

Anomaly based (also called behavior based) intrusion detection systems are based

on profiling normal program behavior and do not rely on definitions for what is malicious. The core principle on which such systems work is that a program under attack behaves differently from normal. Typically, the IDS is subject through a training period during which it learns what is normal for the given program. This unfortunately means that if the training data includes attack behavior, these attacks will be considered part of normal, and never get detected. Anomaly-based IDSs are only as good as their training data, learning algorithm and chosen representation of normal. Anomaly-based systems have the big advantage of detecting zero day attacks.

2.6.4 Anomaly detection approaches

Host-based anomaly intrusion detection was popularized by Forrest et al. [FHSL96]. In that work, normal program behaviour is defined as the observed sequences of system calls seen during training. When the system is in use, only those system call sequences found in the normal profile are allowed to be invoked. The normal profile can be thought of as a security policy specifying which system call sequences, or more implicitly, the control-flow paths of program execution.

Similar systems have been used with features such as instruction branches [BAP+03], CORBA messages [SMS99], and Java methods [IF02]. Some of the recent efforts described in the previous section have looked at anomaly intrusion detection in the context of web applications. Kruegel and Vigna's work looked at statistical anomaly detection of parameters contained in HTTP requests and relied heavily on web access logs [KV03]. Robertson et al. later built on that work to categorize alerts and remove false positives [RVKK06]. This method is very similar to their previous work on system call parameters [KMVV03]. The majority of web application IDSs focus on protecting specific web applications on servers. However, there is some work on the client level. Ismail et al.'s web-proxy based IDS [IEKY04] had no learning componenet, but was based on intrusion detection principles. Noxes [KKVJ06] can be considered an anomaly based approach when its training feature is enabled. However, unlike other systems in the past, JaSPIn has a learning component that automatically infers stable JavaScript policies.

Anomaly intrusion detection is usually not considered policy inference because the

normal profile is not a perfect reflection of security policy. In the system call example, the normal profile does not contain all sequences that are seen during normal runs, creating false positives. Also, because it usually focuses on only one or a few features of execution, attacks that do not alter those features can succeed. However, these flaws are often found in manually specified policies as well. There are well-defined policies for JavaScript in web-browsers, but they are not specific to particular web-applications or web-documents. Also, XSS attacks do not violate these policies in any way. Adding web-application and web-document specific policies is a potential solution to XSS attacks.

Our policy inference engine is an anomaly-based intrusion detection system with a learning component. As seen previously, anomaly based systems are highly effective in detecting attacks or deviations from normal. Thus, they are ideally suited for the web due to the variant nature of attacks, and it's ever evolving nature.

2.7 Summary

JavaScript based attacks are tricky to detect due to their varied input points, masquerading techniques and increased sophistication. The main difference between web pages developed today and those developed a couple of years back, is the amount of JavaScript used to animate the pages and the scope of the modifications that these scripts apply to the web pages after they have been downloaded from the server to the user's browser. AJAX is used extensively in Web 2.0 applications. The central underlying technology of AJAX is a JavaScript API called XmlHttpRequest[W3C07] (XHR) which is available across many browsers. XHR provides a flexible mechanism for sending HTTP requests; and almost any arbitrary request can be sent in the background.

Different techniques such as input filtering, output filtering, taint analysis, web proxies, static code analysis, runtime analysis, anomaly detection and have been applied to detect and/or prevent cross-site attacks.

False positives, false negatives, attack coverage, ease of implementation and ease of use are important considerations when selecting a defense strategy. Comparison of different approaches based on published information is difficult because they contain

Additional efforts	Approaches
Policy/Rules specification	[SS02a], [Inc02], [Min05], [XBS06], [JKK06], [IEKY04]
Application programming	[HYH+04], [Cor06], [JB07], [WS08], [JSH07]
Client-side configuration	[KV03], [IEKY04], [JSH07], [UELX07], [JB07]
Data training	[KV03]
User environment (modified browser)	[VNJ+07]
Language extension	[RDW+07], [YCIS07]
Maintaining black/white lists	[Min05], [XBS06], [JKK06], [KV03], [VNJ+07], [JSH07]
Server-side configuration	[SS02a], [Inc02], [KKP03], [Min05]

Table 2.5: Additional effort to deploy different XSS defenses

varying levels of detail.

Input filtering techniques can generate false positives when input signatures are too restrictive. On the other hand, they can miss a range of attacks if the signatures are too loose. Output filtering and encoding are not only difficult to implement, they are also prone to usability issues. Also incorrect specification of the trusted input source or input string or insufficient signatures can generate false positives. Tainted analysis based detection techniques use either whitelists or blacklists. Both of these approaches are prone to false positives when the list is inaccurate or incomplete. Also, taint based techniques are unable to detect JavaScript attacks where there is no flow of data.

Source code analysis techniques are incomplete due to limitations in implementations that do not guarantee 100 percent coverage. The sheer volume of results and false positives (as high as 50 percent) can be time consuming for the web developer to address. User-specified policy based systems suffer from usability issues. Also, incorrect policy descriptions could lead to either a high false positive or a high false negative rate. Anomaly detection based approaches have proven to be effective in detecting a wide range of attacks. The false negative rate is typically low, while the false positive rate is dependent on the accuracy of the training data. False positive rates are high when the training data is not enough to comprehensively represent normal behavior.

Table 2.5 gives an overview of the additional effort required to implement different defense strategies. Solutions that require modifications to existing applications and additional effort in writing accurate policies have slimmer chances of being widely accepted.

JaSPIn is a client-based solution that requires no change to existing web applications. Also, we have been able to establish that the automatic policies created by JaSPIn are accurate enough to detect attacks while keeping the false positive rate low. More discussion on the same is available in Chapter 5.

We are not aware of any other implementation of inferred JavaScript policies. A related idea has been implemented by Jim et. Al [JSH07], where the website needs to specify which scripts are approved for execution and the browser filters the rest. In comparison to JaSPIn, BEEP lacks the learning component. Also, BEEP not only requires browser modifications, but also modification to every single web page. This is a tremendous amount of effort, and impractical. Also, they make no mention of their false positive rates, while JaSPIn is able to keep its false positive rate low, as explained in Section 5.4

Another similar idea has been discussed in the Mozilla forums: Markham proposes communicating some policies on scripts from web site to browser in an HTTP header. In comparison to our work, his selection of policies is fixed, e.g., 'only scripts in the header are allowed to execute', and do not seem to include policies that could, allow some event handlers in a page to execute, while preventing others.

Hallaraker and Vigna [HV05] propose an auditing mechanism for JavaScript somewhat related to our work, and provide three examples of attacks that can be detected. Their auditing code records method calls, and their arguments. The paper seems to suggest that their initial implementation had a lot of overhead, and was restricted to extremely simple types of XSS only.

Chapter 3

Modeling JavaScript Methods

Different techniques such as input filtering, output filtering, taint analysis, web proxies, static code analysis, runtime analysis and intrusion detection have been applied to detect and/or prevent cross-site attacks. These approaches have only had partial success as discussed in Chapter 2. JavaScript based attacks are tricky to detect due to their varied input points, masquerading techniques and increase sophistication. We propose anomaly detection based on monitoring JavaScript code execution to solve this problem. Anomaly based systems have the advantage of being able to detect previously unknown attacks without requiring had generated policies. This chapter explains the method used by JaSPIn to profile web pages. The following sections elaborate on how anomaly detection can be used to detect JavaScript based attacks.

3.1 Overview

Anomaly-detection algorithms work by constructing models of normal behavior and subsequently checking observed behavior against these models for any significant variations that may hint at malicious behavior. The success of an anomaly detection algorithm depends on the choice of an accurate behavior model. Anomaly based IDSs have been proposed in the past to monitor the system calls made by a process, in an effort to detect deviation from a known profile of system calls for the program. Such IDSs have been used in many scenarios, such as host-based intrusion detection systems (e.g., [FHSL96],[Pro03],[Wag99],[WD01]) and sandboxing and confinement systems (e.g., [PFH03]) In system-call-based anomaly detection, the intrusion detection system maintains state per process monitored and upon receiving a system call from that process (and possibly deriving other information), updates this state or detects an anomaly.

There are several questions that have to be answered when designing an anomaly

based intrusion detection system: where to integrate the intrusion detection system, what should be monitored, and how should the information be represented. We propose a host-based anomaly detection system using anomalies recorded in the browser to detect cross-site attacks. Behavior modeled by our anomaly detection system needs to be automatic, complete and accurate. For this, we propose creating a "'profile"' of every web page visited by the user. A "'profile"' of a web page is a model of the normal behavior of a web page. Any deviations from such established profiles are interpreted as attacks.

The goal of our anomaly detection system is to detect JavaScript based web attacks. A JavaScript based attack executes previously unseen JavaScript code on a web page. Creating profiles that define normal JavaScript code execution can be used to detect code that was not previously executed. As seen in Chapter 2, JavaScript is an object-based language. Client side JavaScript uses many pre-defined and user-defined objects. The JavaScript execution environment in the browser registers a large number of function callbacks on initialization. These hooks are used when scripts try to access a specific property or method that are not native in the engine. To ensure completeness of our anomaly detection system, all these calls must be intercepted and logged. The profile of a web page visited by the user is created by logging all method calls and property getters and setters in the JavaScript interpreter. To improve the accuracy of our intrusion detection system, we also record the order in which these method invocations occur.

3.2 Threat Model

There are a wide range of security problems prevalent on the web today. No single computer or network security implementation is effective against every type of attack. By using a few different security products together, it is possible to eliminate the vast majority of attacks. This concept of having multiple defenses working together to protect a system or user is termed 'layered defense'. In such a defense, each solution lives up to a threat model. A threat model describes the assumptions and factors considered while making a solution. It also describes the problems that are addressed by the solution. This section elaborates on the type of attacks JaSPIn is built to

detect and the underlying assumptions and general requirements of JaSPIn.

The goal of JaSPIn is to detect malicious JavaScript that is not normally part of a web page trusted by the user. Being an anomaly detection based approach, the central premise of JaSPIn is that intrusive activity is a subset of anomalous activity. In the ideal case, the set of anomalous activities will be the same as the set of intrusive activities. However, as explained in the previous chapter, this is not always true. Our goal, is thus to reduce the number of reported non-intrusive activities while being able to detect all intrusive activities.

Although we assume that an attack will generate an anomalous sequence by calling a different set of functions in a different sequence of calls, it is possible for the attacker to mimic normal behavior by calling functions in the same sequence as the most common paths for all users. This type of attack against anomaly detection systems is termed a mimicry attack. Mimicry attacks against JaSPIn are discussed in details in Chapter 6.

We assume that the attacker does not have control over the user's computer and does not have the ability to spoof JaSPIn or to create or control JaSPIn generated policy files. The attacker is also assumed to have no internal control over the trusted site. JaSPIn does assume that the attacker controls arbitrary web servers and some of the content on legitimate servers (but not the source code directly).

It has to be clarified that although JavaScript can also be used to exploit vulnerabilities in the browser itself, this thesis does not look at that class of attacks. These attacks are independent of the website and not cross site. The only case, in which XSS could be related to browser vulnerabilities, is when exploitation of these browser vulnerabilities requires powers available only to a particular target site.

Thus, this thesis does not discuss situations where an attacker compromises a web browser to circumvent profile checks or modifies the profile of a web page. Further, we do not address the problem of users visiting already compromised malicious web sites that do not have a stable profile. Instead we focus on the problem of protecting a protecting the user against cross-site attacks on sites that the user frequently visits such as her bank's web site, email web client or news web sites.

3.3 Description

Our system observes JavaScript code as and when it is interpreted in the browser. Although any JavaScript downloaded by a web page may have the potential to do harm, it has to be actually running to realise that potential.

3.3.1 Choice of Algorithm

A method for classifying JavaScript behavior based on method call invocations could record and measure method invocations in many ways. It could compare the timings of different calls, or their relative frequencies. It could analyze arguments to specific JavaScript functions, or could only look at a subset of all possible method invocations.

We have chosen to create the profile of a web page by recording all the JavaScript method calls, including property setters and getters made by the web page. Recording only the name of the method call provides completeness, but not accuracy. A malicious method executed by the attacker's code could have the same name as that found in the normal profile of a web page. To improve the accuracy of our model and increase the effort required by an attacker to mimic normal method call behavior, we record the relative order in which these method invocations happen. We decided to use this simple approach in our current implementation.

A modeling algorithm implemented in the browser needs to be able to capture a large number of method invocations in a short time. Also, this information on method invocation needs to be stored in a compact way that converges to a fixed state profiling the web page. Once a stable profile has been created, the algorithm needs to be able to detect anomalous events efficiently.

Previous work by Hofmeyr [HFS98] and Somayaji [Som02] have used two techniques, both of which use a fixed-length window to partition a process's system calls into sequences. Window size is defined as the length of the subsequence of a call trace. It is used as the basic unit for modeling program or process behavior. The most straight forward technique used in anomaly detection systems is called the sequence method. In this method the profile of a program's behavior consists of the entire set of sequences produced by that program. With the other technique, known as the lookahead pair method, the pairs formed by the current and a past system

call are stored in the program's profile. Look ahead pairs are less suitable in our case due to the much large alphabet size in generating profiles for web pages. The sequence method has worked surprisingly well in comparison to other, more sophisticated algorithms. Experiments in the past have showed that the sequence method was almost as accurate as the best algorithm in any given test, while being much less computationally expensive [WFP99].

3.3.2 Profile Generation

In our current work, we record the sequence of method calls in the JavaScript interpreter that precede the current call. Depending on the window size (n), the profile contains the series of calls (n-1) that precede it. The entire set of sequences of all method invocations called during the user browsing a web page form the profile for that given page. This includes constructors, property getters, setters and garbage collection methods that can be traced in the JavaScript interpreter.

The algorithm used to build the normal profile is simple. We track all method invocations including property getters and setters generated by a particular web page, and build up a profile of all unique sequences of a given length that occurred during the trace. Each web page of interest has a different profile, which is specific to the functionality of the JavaScript embedded in the page, version and configuration of the browser, local administrative policies, and usage patterns. Once a stable profile is constructed for a given web page, the profile can be used to monitor the ongoing behavior of the JavaScript code invoked by that web page.

The construction of the normal profile is best illustrated with an example. Suppose we observe the following trace of method invocations:

> document, getElementsByClass, className, length , document, getElementsByClass, length

Note that in the example above, the documet object is also included as a part of the trace. This is because of a call to the new() method to create a new document object to work with.

We slide a window of size w across the trace, recording each unique sequence of length that is encountered. For example, if w=3 we get the unique sequences:

document, getElementsByClass, className
getElementsByClass, className, length
className, length , document
length , document, getElementsByClass
document, getElementsByClass, length

For efficiency, each unique method call is stored in a map file and the corresponding index is logged.

Somayaji presents a more formal description of this algorithm [Som02]. We have modified the same to suit web profile generation.

Given,

C = the alphabet of all possible JavaScript method invocations made by a particular page
c = $|C|$
T = $t_1; t_2, \ldots, t_n \mid t_i \in C$ (*The trace of calls*)
τ = the length of T
w = the window size where $1 \leq w \leq n$
P = the set of patterns associated with T and w (profile)
n = number of web pages in the website

$$
\begin{aligned}
P_{webpage} = \{ & < s_i, s_{i+1}, \ldots, s_j > : si, si+1, \ldots, sj \in C \\
& 1 \leq i, j \leq \tau, \\
& j - i + 1 = w, \\
& s_i = t_i, \\
& s_i + 1 = t_i + 1, \\
& \ldots \\
& s_j = t_j \}
\end{aligned}
$$

The profile for a given web page is the set of all sequences of length w of JavaScript method invocations invoked during the training session from the trace of calls T.

3.3.3 Profiling a complete web site

To generate the profile for a given website, we look for similarity in profiles between two web pages in the same domain, and group them together. We compare the profile of a newly visited web page with other web pages in that domain at the same level, and if no profile has been generated at the same level, it is compared to a profile for a page one level above, if it exists.

A profile to be a set of sequences and thus, we use set theory to explain how profiles are merged.

Suppose we need to find the if the profile for `\tauhttp://www.mydomain.com/sublevel1/page1.html` can be merged.

1. Check if a profile exists for `http://www.mydomain.com/sublevel1/somepage.html`
   If yes, check if merge is possible.
   If merge, the union of these two pages is renamed to create a profile for `http://www.mydomain.com/sublevel1/`. Quit.
   If merge is not possible, Quit. Keep profile separate.

2. Check if a profile exists for `http://www.mydomain.com/sublevel1/`
   If yes, check if merge is possible.
   If merge, add new sequences to profile `http://www.mydomain.com/sublevel1/`. Quit.
   If merge is not possible, Quit. Keep profile separate.

3. Check if a profile exists for `http://www.mydomain.com/`
   If yes, check if merge is possible.
   If merge, add new sequences to profile `http://www.mydomain.com/` Quit.
   If merge is not possible, Quit. Keep profile separate.

Profile similarity is computed for two profiles from different web pages of the same web site, for example tech.yahoo.com and music.yahoo.com.

We measure profile similarity by looking at the number of sequences that appear in the intersection of the two profiles.

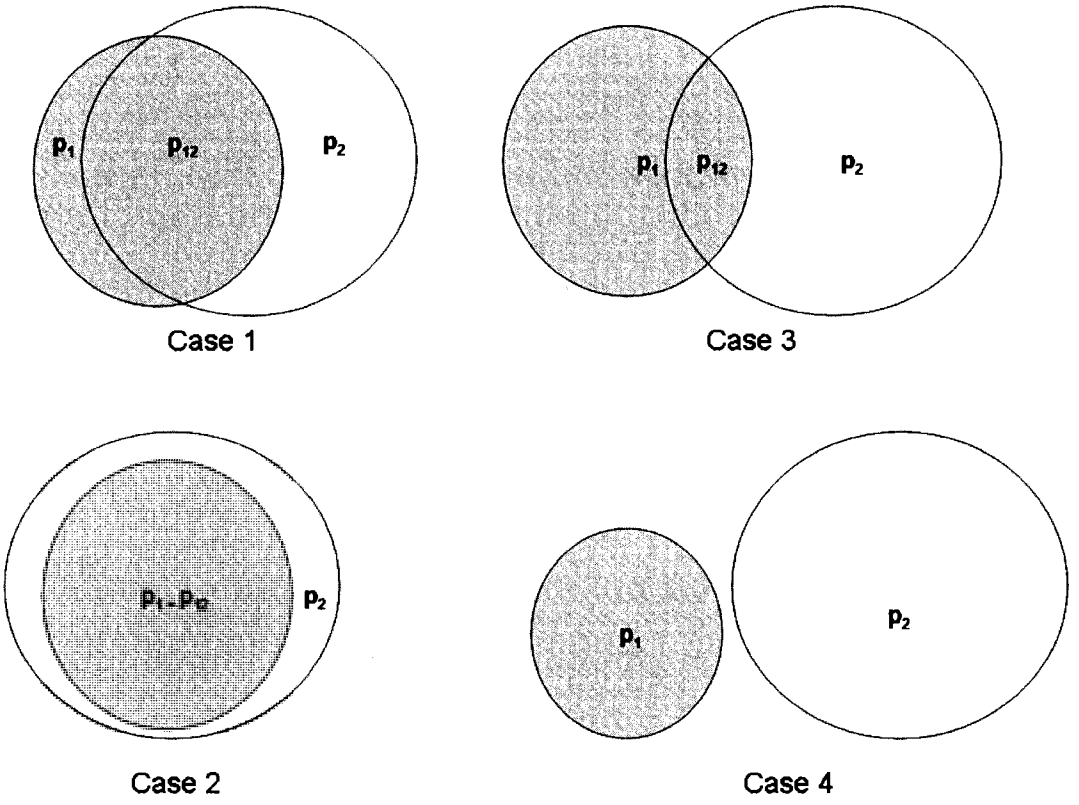

Figure 3.1: Finding proximity similarity between two web pages in the same domain

In Figure 3.1,

p_1 = Sequences in the first web page p_2 = Sequences in the second web page p_{12} = Sequences that appear in both web pages

For a configurable proximity threshold t,

Case 1, Case 2: $p_{12} \geq$ t : Merge profiles

Case 3, Case 4: $p_{12} <$ t : Do not merge profiles

Thus, if two web pages with similar profiles need to be grouped,

$$P_{webpage12} = s_i, s_{i+1}, ..., s_j \; U \; s'_i, s'_{i+1}, ..., s'_j$$

where

$$s = sequence\ in\ P_{webpage1}$$
$$s' = sequence\ in\ P_{webpage2}$$

3.3.4 Detecting Anomalous Behavior

Once we have a profile of normal behavior, we use the same method that we used to generate the profile to check for new traces of behavior. We look at all overlapping sequences for the selected window size in the new trace and determine if they are represented in the profile of a web page. Sequences that do not occur in the normal profile are considered to be mismatches. To detect an intrusion, at least one of the sequences of JavaScript method calls generated by the intrusion must be classified as anomalous. When a new call is added, up to w anomalous sequences are produced, one for each possible position of the new call.

The most common form of cross-site attacks involves accessing the document.cookie method in the attacker's script. While accessing the cookie of a web page, the JavaScript program first accesses the document object which is a property of the window object. The cookie object is a property of the document object. Calls to these property getters are traced by our intrusion detection system as method calls in the interpreter. More advanced cross-site attacks such as DNS pinning and port scanning also cause anomalous behavior and are explained in detail in Chapter 5.

3.4 Analysis

JaSPIn is a client side tool that uses the sequence method to infer JavaScript profiles. JaSPIn's profiles are inferred from behavior. The fundamental principle of such a system is that most attacks will cause a deviation from normal behavior. Normal behavior is automatically profiled by JaSPIn. Learning based algorithms reduce the user's workload. User devised policies could be incomplete, and any system that relies on the user is only as good as the user who sets it up. Another parallel approach could be for the programmer to set up these policies as he is the one with complete knowledge about the function calls. However, this method is not only error-prone due to reliability on the programmer, but is also more prone to mimicry attacks.

As with any empirically derived model of normal behavior, such automatic policy generation comes with the risk of imperfect detection, that is, false positives and false negatives. False positives can be caused by erratic changes in the user's browsing

patterns, or updates to the JavaScript functions used on a web page. A false negative is a situation where an intrusion is really happening, but the IDS does not catch it. In our case, for an attack to actually happen, it does require the attacker's script to execute. This can be caught as a violation of normal behavior. However, mimicry attacks against JaSPIn are possible and this is discussed in Chapter 6.

The present version of JaSPIn does not implement any response mechanisms. Although detection is done real time, and anomalies identified before an attackers' code can actually execute, currently we have not implemented any way to mitigate these attacks. This research aims only to detect JavaScript based attacks against web applications. Automatic mitigation of these attacks is important, and discussed in Chapter 6.

3.5 Summary

This chapter described our profile generation methodology, which is based on an anomaly intrusion-detection system for web applications to detect JavaScript based attacks. The scope of this research is limited to JavaScript based cross site attacks, and does not aim to detect any browser vulnerabilities. Also, JaSPIn's policies do not protect against all web application attacks. While XSS acts as the means for many other attacks including, SQL injection and remote code injection, these attacks are also possible by exploiting other server or application vulnerabilities. JaSPIn is only one layer in protecting against web based attacks. Additional, defense-in-depth methods to add extra layers of protection are needed.

Chapter 4

Implementation

JaSPIn works by profiling the list of JavaScript function calls made by a given website to create profiles for that particular site. Our solution works at the client, and thus requires the browser to be able to provide us with information on the script being executed. This chapter elaborates on the changes we did to the Mozilla Firefox browser. Our example implementation enables seeing JaSPIn in action and allows us to investigate the efficiency of this method. The first part of this chapter explains the architecture of the browser. We then explain how JaSPIn has been implemented by modifying Spidermonkey and integrating our anomaly detection system in it. We also give some details on the implementation and working of a browser extension that can be used to talk to the internal policy engine. The last part of this chapter explains how JaSPIn works with some examples. We have extended the Mozilla Firefox 2.0pre web browser [W3C04] from the Mozilla Foundation. The modified browser was successfully built with help of the build documentation on a Fedora Core v4 Linux system.

4.1 Understanding Mozilla Firefox

Firefox is built on top of Mozilla's application platform and reusable components.

Figure 4.1 [Fou08] gives an overall view of the browser. The JavaScript Interpreter, SpiderMonkey is of particular interest to us. Mozilla is a large and modular software written in C, C++, and JavaScript. The use of several technologies allows Mozilla projects can be developed independently. The main mechanism that supports the integration of the different components is the Cross-Platform Component Object Model (XPCOM) which is similar to Microsoft's Component Object Model (COM). Other technologies used are XPConnect and the Cross-Platform Interface Definition Language (XPIDL). The following sections explain each of these in detail

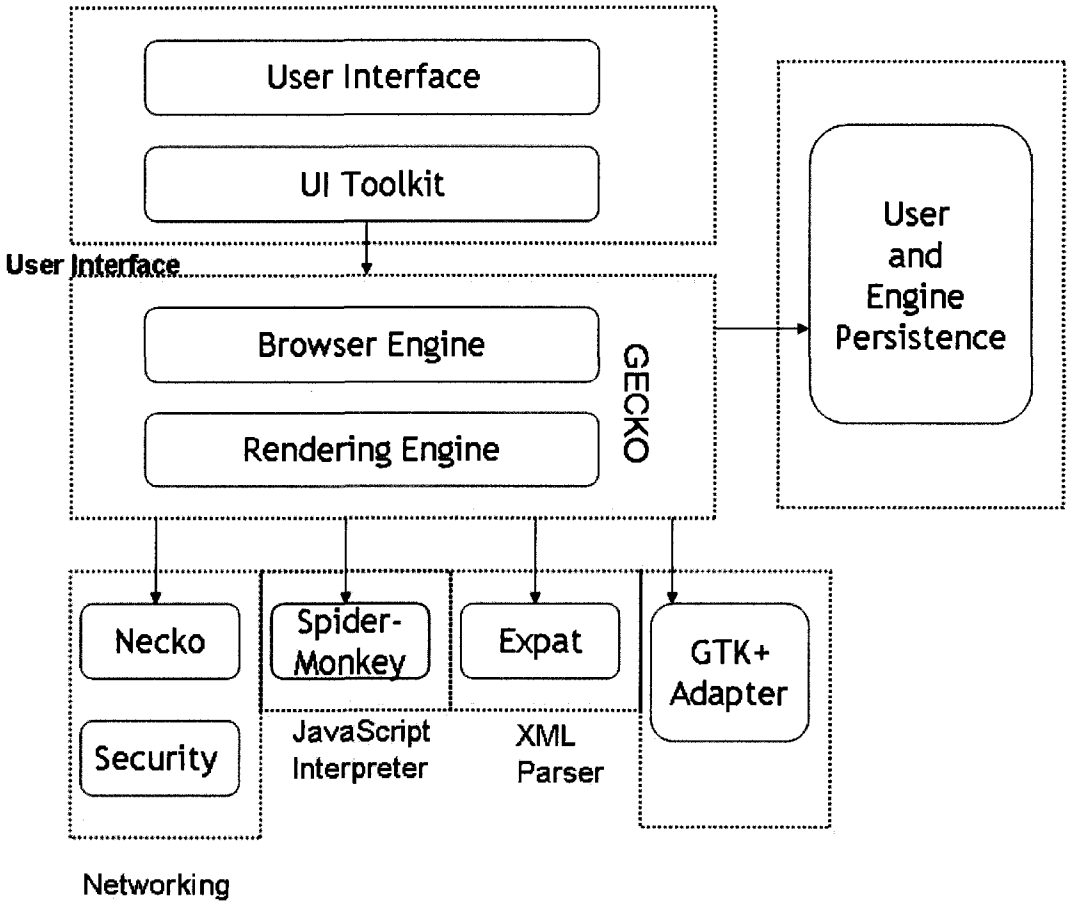

Figure 4.1: Architecture of Mozilla

4.1.1 SpiderMonkey

SpiderMonkey [Cor08] is the code-name for the implementation of the JavaScript engine embedded in Mozilla. It is a stand-alone JavaScript engine that parses, compiles, and executes JavaScript code. The engine conforms with the ECMAScript standard [Int99], which defines built-in data types. In addition ECMAScript defines a collection of built-in objects which include the Global object, the Object object, the Function object, the Number object, the Math object, the Date object, the RegExp object and some Error objects. Any applicationthat embeds SpiderMonkey can define

its own application-specific objects in addition to the built-in objects. In a browser like Firefox, the application-specific objects are responsible for providing access to the Document Object Model (DOM) [W3C04] from within the JavaScript engine. The DOM contains an object-instance hierarchy that models the browser window and some browser window information. It also contains an object-instance hierarchy of elements of an HTML document, which is created when the document is loaded into the browser. For example, some of the objects made accessible by the DOM are the window object, the document object, the navigator object, and the location object. The window object is the global object from which all other objects inherit. The HTML elements of the current document are represented bu the document object. Each object has a number of properties which can either be a built-in type, an object, or a method. An example of this is the href property accessed using the expression document.referer.href. A JavaScript program first accesses the document object which is a property of the window object. The referer object is a property of the document object, and href is a property of the referer object.

SpiderMonkey exposes a public API that applications can use to compile and execute scripts, instantiate host objects, and define properties. The engine does not provide any security per se, and all mechanisms to provide access control and safety must be implemented in the embedding application, e.g., the web browser.

JavaScript and C++ interact with each other in the Mozilla source code. C++ is a compiled language, while JavaScript is an interpreted language. When the browser is started, the C/C++ components start first. But in an early stage, XPConnect is initialized and enables the use of interpreted JavaScript at runtime. A Mozilla browser distribution consists of both compiled C++ and uncompiled JavaScript files. As explained in Chapter 2, JavaScript code from a web page is executed in its sand box, and does not have any access to Mozilla's internal objects. It can however access objects that are exposed by the DOM. JavaScript is mostly used in those areas of the source code that care for user interface events. This also means that the SpiderMonkey interpreter executes both scripts on behalf of a downloaded web page and scripts that are part of the "native" code of the Mozilla browser. Thus, when mnitoring JavaScript method invocations, we need to be able to differentiate between

the two.

4.1.2 XPCOM

XPCOM provides a platform and language independent modular framework that enables a software project to be broken up into smaller modularized pieces that are integrated at runtime, and separates the implementation of an object from its interface. The basic idea is that related functionality is gathered in one entity, called a component or a module. The component implements one or more interfaces through which other components can access its functionality. An interface consists of one or more methods and variables. Each component has a unique classID and contractID that describe the component. In addition, each interface the component implements has a unique InterfaceID which must be specified before accessing the component. The component manager keeps track of all the components in the system, and is responsible for finding the correct component when a contractID or classID is specified.

4.1.3 XPConnect

XPConnect is the layer between XPCOM AND javaSCript and is responsible for them to work together. It provides a transparent interface to XPCOM objects and allows JavaScript objects to interact with XPCOM objects. JavaSCript objects can also implement XPCOM compliant interfaces. XConnect bridges the gap between the two. Such an approach was adopted to keep the various components seperate and provide a transparent mechanism for object interaction between SpiderMonkey and XPCOM.

4.2 System Architecture

The first step in designing JaSPIn was to determine where our profiling mechanism would be built and what it would do. Also, profiles need to be represented in a precise and compact way. The anomaly detection engine needed to be either in XPConnect, which is used for forwarding most calls directed to the DOM or directly in SpiderMonkey. Since not all calls go through XPConnect, and in our case it

is necessary that we log every single method invocation, JaSPIn is integrated in SpiderMonkey itself. SpiderMonkey however does not have explicit knowledge of whether the script being executed is part of the web page or the JavaScript engine. This distinction is possible in XPConnect. In a JaSPIn enabled browser, modifications to the DOMClassInfo class have been made to pass this information to SpiderMonkey from XPConnect.

One key challenge in doing this is to be able to differentiate between "native" scripts that execute on behalf of the browser, and scripts that are downloaded as part of HTML pages. Since "native" JavaScript code executes with all privileges set, these scripts can perform any operation that is allowed to the browser program itself and should not be audited. Another important choice was to decide how much information is required to be logged. It was unclear in the beginning if only the sequence of method invocations would suffice, or the arguments to those calls mattered. From our initial experiments, we were able to determine that for our anomaly detection system; a stable normal profile for a given website is possible just based on the sequence of method calls. However, in order to reduce the number of false positives, we track which top level event such as an onclick or onmouseover gets triggered before a set of sequences.

Each web site has three files associated with it. The first time a website is visited a new map file containing all the method calls made by it is created. The map file has only function names recorded in it. Each function begins in a new line, and the corresponding line number is used for tracking their order in the log file. The log file records the order of method invocations. The profile of a given website is created by generating sequences of these calls with a window size specified by the user. Profiles are saved for every web site in a .prof file. The log file is deleted once the corresponding profile has been generated.

An important decision that we had to make was to determine if we would profile every web page or every web site. This is a tricky question - since profiling every web page would decrease our false positive rate, also decreasing JaSPIn's ability to detect attacks. For example, if we were to profile every web page separately then, `http://www.cnn.com/2008/POLITICS/03/17/florida.primary.decision/index.html` and

`http://www.cnn.com/2008/US/weather/03/17/atlanta.tornado/index.html` would each have their own profile. This means that every time the user visits cnn.com, new profiles will be created for every news item thereby making it almost impossible to detect attacks. When we look at the profiles of both the URLs above they are the same. Also, logically, we can see that they are indeed the same page. On the other hand, if we were to profile the entire website as a single entity, there will be a high rate of false positives, as the profile of cnn.com/weather is different from the news pages. Our approach is thus to group profiles of similar web pages from a domain automatically, thus creating separate profiles for the weather section, and a single profile for the news pages. Similarly, we needed to decide of it would be significant to consider query strings at all. From our observations, we found that the inclusion of query strings to separate profiles did not provide any added benefit. Hence, the current version of JaSPIn ignores query strings in both profile creation, and policy verification. We only observe pages that use JavaScript. The lookup of function names in the profile is currently implemented as a linear search through the file. The profile file format is simply a list of sequences with the specified window size. A summary of the algorithm is presented in pseudo code 1 . This code has two parts - the first set of functions is executed during profile generation, and the second set during attack detection. Profile generation is enabled if the user opts for this in JaSPIn's user interface, and if the stability threshold has not yet been reached. Given a stability threshold of "'N"', a profile is considered stable if no new sequences are generated for "'N"' continuous visits to the same web page. The default stability threshold selected is three. This is explained in detail in 5.2.1.

```
Input: Map file,Profile, Current Method Call Name
Output: Alert if anomaly detected
if jaspin_status == ON then
    foreach Method Call m do
        if enable_profile then
            if isnew(profile,m) then
                add_function(Map,Profile,m);
            end
            get_sequence(m);
            if NOT profile_test(Profile,get_sequence(m)) then
                profile_add(Profile,get_sequence(m));
            end
        end
        else
            if isnew(profile,m) then
                signal(Anomaly);
            end
            get_sequence(m);
            if NOT profile_test(Profile,get_sequence(m)) then
                signal(Anomaly);
                sequence_print(get_sequence(m));
            end
        end
    end
end
```

Algorithm 1: The JaSPIn algorithm applied on every method invocation

4.3 Implementation

The following sections will give some details on our code modifications to SpiderMonkey, building of the extension to manage JaSPIn and the interaction between the two. For our example implementation, Mozilla Firefox browser 1.5.0.6 was modified. The source code for Firefox can be downloaded from

ftp://ftp.mozilla.org/pub/mozilla.org/firefox/releases/
It was built on Linux version 2.6.11 (gcc version 4.0.0). Since JaSPIn requires changes to SpiderMonkey, a fresh installer for Firefox with these modifications was created for both Linux and Windows platforms. The extension comes installed on our version of Firefox, but can be updated or deleted like any other Firefox extension. When installed on an unmodified version of Firefox, the extension simple does not do anything. The Mozilla build system, like the rest of the Mozilla codebase, is cross-platform. It uses traditional unix-style autoconf and make tools to build the various applications. A .mozconfig file, which sources from the Firefox default mozconfig file is placed in the source directory (mozilla/.mozconfig). The following command is then used to build Firefox. make -f client.mk build

4.3.1 Changes to SpiderMonkey

The source for SpiderMonkey is found in the js/src/ folder under the main source directory of Mozilla. Both the LiveConnect and XPConnect interfaces are also present under this directory. All the classes are implemented in C and C++. The core classes that we modified are all written in C. JS modules declare and implement the JavaScript compiler, interpreter, decompiler, GC and atom manager, and standard classes. The compiler consists of a recursive-descent parser and a random-logic rather than table-driven lexical scanner. Semantic and lexical feedback are used to disambiguate hard cases such as missing semicolons, assignable expressions ("lvalues" in C parlance), etc. The parser generates bytecode as it parses, using fixup lists for downward branches and code buffering and rewriting for exceptional cases such as for loops. It attempts no error recovery. The interpreter executes the bytecode of top-level scripts, and calls itself indirectly to interpret function bodies (which are also scripts). All state associated with an interpreter instance is passed through formal parameters to the interpreter entry point; most implicit state is collected in a type named JSContext. Therefore, all API and almost all other functions in JSRef take a JSContext pointer as their first argument. JaSPIn initiates itself whenever a new context is created. This is achieved by modifying the js_NewContext function in jscontext.c. By doing so, parallel logging is enabled when multiple tabs are opened, or

```
typedef struct sequence_profile {
    profile_t           funcs;
    sys_call_map_t*     map;
    int                 window;
    GHashTable*         states;
} sequence_profile_t;
```

Figure 4.2: Struct definition for storing each profile

multiple browsers are used. Our logging mechanism is implemented in jslog.c, which is called from the other source files.

Modifications were done to the functions that interpret script method invocations. The previous function call, context and web site are remembered till a change occurs. Files modified include js.c, jsfun.c, jsparse.c and jsapi.c, all in the js/src directory.

The primary aim of finding each and every function call is to be able to sequence them to create a set of sequences that can be either used as part of the profile during training or for comparison during detection. JaSPIn integrates sequence generation, and sequence comparison classes into the src directory of SpiderMonkey. Figure 4.2 shows the information stored in the sequence_profile structure.

Each profile contains the list of functions as a pointer to the map list and the window size, which can be varied in the JaSPIn extension. Profiles are saved in memory till the user moves away from a particular web page. It was challenging to determine the URL of a web page from within SpiderMonkey, and we needed to have interface classes via XPConnect to the outside to determine this information.

4.3.2 Browser Extension

An intrusion detection system will only be used if it is user-friendly, and gives the user a sense of control. Since all the code changes for JaSPIn are done deep inside SpiderMonkey, we needed to have an interface to allow the user to interact with the same. JaSPIn's user interface is built as a standard extension for Firefox. Firefox extensions allow the application to be customized to fit the personal needs of each user if they need additional features, while keeping the applications small to download.

Thus, extensions can be made publicly available for download. If the JaSPIn extension is installed on a system that does not have the SpiderMonkey modifications, it simply alerts the user and does nothing. The JaSPIn extension cannot be initialized without the modified SpiderMonkey core. As with all Firefox extensions, our user interface is written in XUL and JavaScript. XUL is an XML grammar that provides user interface widgets like buttons, menus, toolbars, trees etc. User actions are bound to functionality using JavaScript. The extension can be used either from the Tools menu or by clicking on the icon on the top right. The extension also provides the status of the system at the bottom right corner of the browser.

The extension acts as both a user input and a tool output mechanism. The user is able to enable or disable policy generation or detection. Also, the tolerance levels for false positives can be specified. Window sizes for each sequence in the profile can also be modified. There is also a white list which specifies which websites the user does not want to profile.

It also adds functionality to display the ongoing status of JaSPIn in the status bar, as shown in Figure 4.3. It indicates whether JaSPIn is currently generating profiles or verifying a web page state against its profiles. If anomalies are found, the number of anomalies detected is mentioned in the status bar with a blinking warning icon. When the user clicks on this, further details of the anomalies detected are revealed. If the user classifies the detected anomaly as a false positive, she is given the option to update the profiles.

A snippet of a MAP and Profile file generated by JaSPIn is shown in the Appendix.

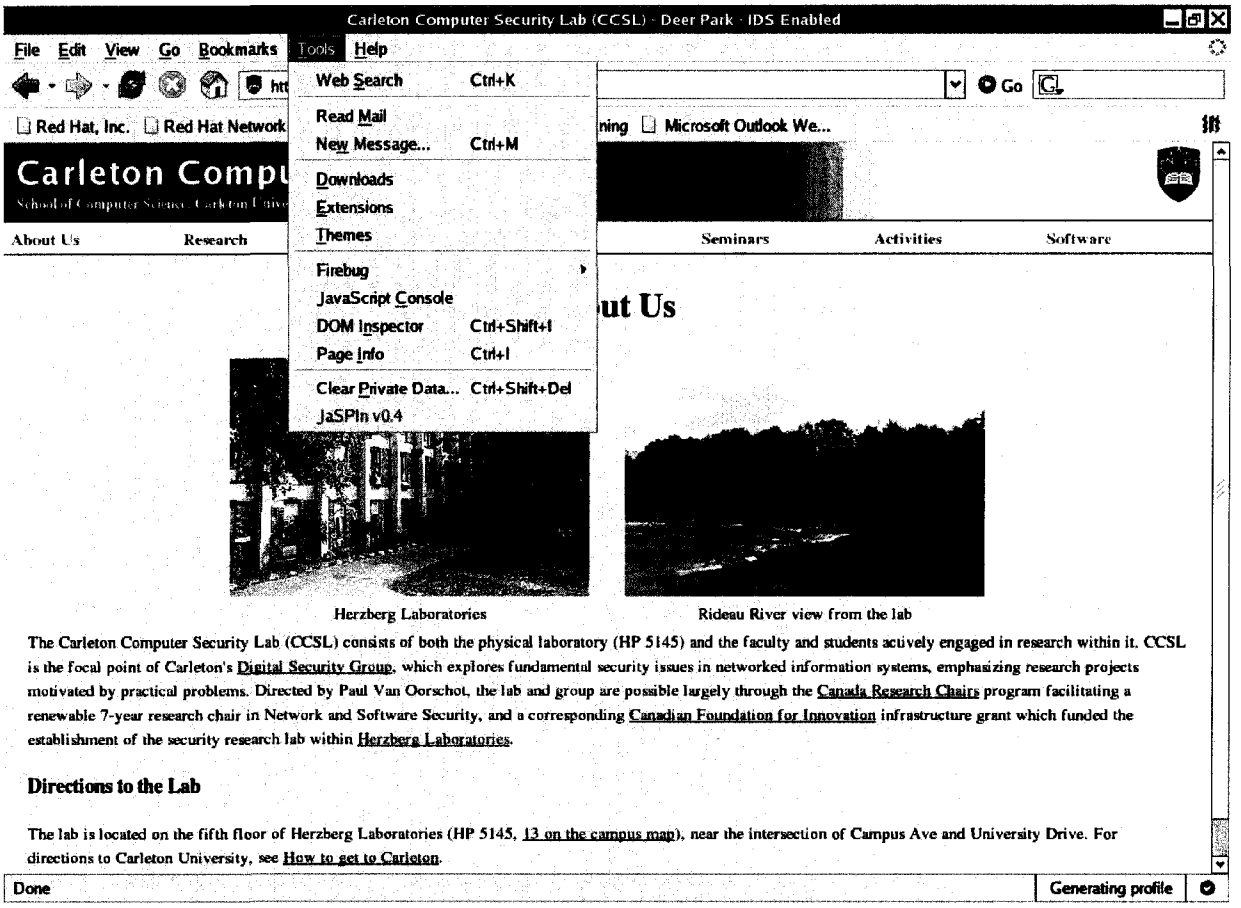

Figure 4.3: JaSPIn extension

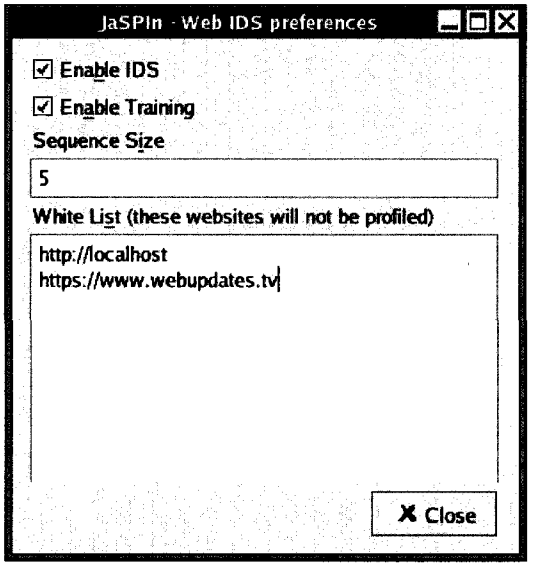

Figure 4.4: Configuring JaSPIn using the extension.

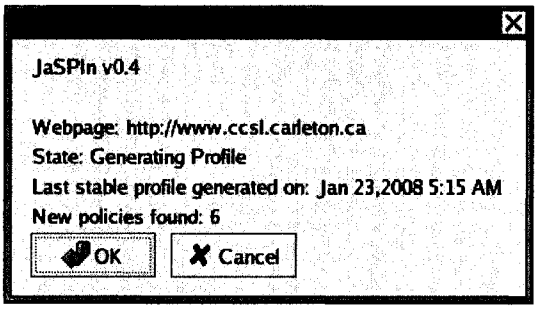

Figure 4.5: Alert while using JaSPIn

Chapter 5

Results

Efficient attack detection is an important performance measure of intrusion detection systems. However, detection rates alone are not an adequate measure of the efficacy of an intrusion detection system because alarms will be ignored if the system produces too many false alarms. Our experiments with JaSPIn measure the probability of false alarms using JaSPIn, its effectiveness in detecting various kinds of cross-site attacks, resource usage and its impact on web page load times.

In this chapter, we present results from our experiments using JaSPIn. We first describe the influence of window size and the stability threshold on the functioning of our anomaly detection system. We then present results from our experiments using JaSPIn to determine profile stability, false positives, profile diversity, overhead, and performance. We also describe the experiments on evaluating the effectiveness of JaSPIn against some common XSS attacks, including advanced cross-site attacks that can evade several other detection models. Table 5.2 outlines the various experiments conducted using JaSPIn. Each of these tests is explained in detail in the following

Type	Test category	No.of tests	Aim	Findings
A	Profile stabilization tests	60	To determine if JavaScript method sequences are repetitive across visits	Stable policies can be generated in about 5-15 visits
B	False Positive Rate	60	Determine number of false positives	As low as 0, decreases with increasing visits
C	Internet web surfing simulation	25	Profile diversity	yahoo.com showed varying profiles for varying runs
D	Overhead	2	Determine space and time requirements	Small profile files, low impact on we page load times
E	Simulated Exploits	59	Attack detection	Common XSS attacks detected
F	Targeted attacks	3	Attack detection	Successful detection

Table 5.2: Summary of tests conducted using JaSPIn to determine its accuracy, performance, overhead and efficiency. Tests (A) to (C) focus on determining the accuracy of our model of normal behavior; (D) analyzes space and time requirements; (E) and (F) test JaSPIn's attack detection efficiency.

sections of this chapter.

5.1 Data Source

Our experiments were conducted by installing JaSPIn on a Linux 2.6.11 Fedora Core 4 computer. Sixty different websites (see Appendix) were profiled and revisited during the course of three months. These websites were not handpicked, and have been profiled during day to day browsing. Although not exhaustive, this list provides a good sample of websites different from each other with respect to their purpose, usage of JavaScript, programming language etc.

Synthetic normal profiles can be created by doing a static or runtime analysis of the code to find method invocation sequences. However, not all sequences in such a profile will be generated in practice, since one cannot determine whether a behavior is frequent, rare or unseen for a particular user, and so one cannot get a true sense of the false positive rate in practice. Our approach is to examine the behavior of JaSPIn by gathering data from real websites. This strategy has some disadvantages. First, the experiments cannot be exactly replicated, because the conditions of the tests cannot be exactly duplicated. Data gathered from live web sites may also be contaminated with actual security violations, potentially making it difficult to distinguish between true positives (genuine attacks) and false positives (other anomalous behavior). Also, such experiments are dependent on the web sites visited. These disadvantages are outweighed, however, by the prospect of discovering how well JaSPIn works in practice on real websites.

5.2 Choice of Parameters

The windows size (refer Section [3.3.1]) and stability threshold (refer Section [??]) used by JaSPIn can be configured by the user through the browser extension. The default value for the window size is 6 and the stability threshold is 3. To better understand the rationale for these values, the following parts discuss how the values of these parameters affect the behavior of JaSPIn.

Website Type	No.of Websites	Methods	Avg. Methods	Num. of visits across window sizes					
				2	4	6	9	15	20
Moderate JavaScript	17	<100	59	4	6.1	5.8	9	9	9
JavaScript Intensive	31	>100	87	6	12	15.2	17	19	21
AJAX intensive	12	>100	110	9.1	11	13.7	16.7	22.4	24

Table 5.3: Effect of varying window sizes on the number of visits to achieve profile stability with a threshold of three

5.2.1 Window Size

JaSPIn attempts to predict whether a sequence under observation is more likely to have been generated by a page's normal script or malicious code. This is done by observing how consistent the sequence is with the sequences in the profile of the web page. As seen in Chapter 3, the length of the sequence is called the window size. JaSPIn uses a fixed window size to build the profile of a web page during training, and compare it against generated sequences seen during testing. The size of the window is configurable as seen in Chapter 4, and is picked a priori.

There is a trade off between using shorter or longer sequences (window sizes). Longer sequences produce more anomalies because any deviation from previously-seen patterns will create new sequences in proportion to the length of the window. If we use longer sequences, we have a lot fewer instances of each subsequence in our data. However, longer sequences provide better attack detection and are more difficult to mimic by malicious code.

In the past, researchers have used window sizes of six [HFS98] and nine [Som02] for system call anomaly detection. JaSPIn was configured to use these values. Additionally, we also tested smaller window sizes of 2 and 4 and larger window sizes of 15 and 20. Table 5.3 shows the effect of varying window sizes on the number of visits required to create a stable profile of different websites with default thresholds (explained in the next section). We have classified the sites visited during this experiment into three categories based on their usage of JavaScript indicated by the number of methods in the map file. As explained in Chapter 4, the map file contains the list of all method invocations made by the web page. We determine the average number of visits required to a web site to define its profile for varying window sizes. Shorter sequences of two and four require the least number of visits. However, these are too small for attack detection, since many JavaScript property setters and getters

Web site type	URL	Visits for a stable profile
Static Content	http://www.ottawahomerepairs.com	-
Dynamic Content	http://www.imdb.com	4
Dynamic Content	http://en.wikipedia.com	12
JavaScript Intensive	http://www.rogers.com	14
JavaScript Intensive	http://www.cbc.ca	21
JavaScript Intensive	http://www.cnn.com	22
AJAX Intensive	http://mail.yahoo.com	11
AJAX Intensive	http://maps.google.com	19
Flash	http://www.fitc.ca	-

Table 5.4: The number of visits for some sample websites for a stable profile to be generated

require four calls (Ex: window.document.forms[0].SquareFeet.value). This sequence can not only be a part of the normal profile of a web page, but also the attacker's script, thus permitting mimicry attack. Longer sequences of fifteen and twenty require larger number of visits and more storage. In our experiments with JaSPIn, we use a window size of six, since it requires the least number of visits to create a stable profile for a web page and is greater than four.

5.2.2 Stability Threshold

The stability threshold (refer Section [??]) of JaSPIn is a configurable option available to the user. It is used to dictate the number of consecutive visits to a web page that produce no new sequences to be added to the web page's profile during the training phase. Once the stability threshold has been reached, JaSPIn switches from training to detection phase for that particular profile. The stability threshold is a parameter dependent on the user's browsing patterns and tolerance to false alarms, and can be better chosen after more rigorous testing. Higher thresholds translate to more training data due to increased visits to the website during training. More training reduces the number of false positives. In our experiments, we have opted to use a low stability threshold of three.

5.3 Profile stabilization

JaSPIn detects intrusions by noticing anomalous sequences of JavaScript method invocations. Hence, it only detects attacks against web applications for which it has a normal profile. This section examines how well JaSPIn can acquire them in practice.

A profile is considered stable if training on the current web page is complete, and attack detection is enabled. In other words, the stability threshold has been reached. Table 5.4 shows the number of visits required to generate a stable profile with a window size of 6 an stability threshold of 3. For tabulation purposes, if the profiles for different pages on a web site are different (as explained in Section 4.2), we provide the total number of unique sequences as the number of sequence for the over all web site. No restrictions were imposed on the browsing patterns in an attempt to keep it real. Consecutive visits to these websites are not necessarily immediate and these results were collected over a period of six months. Although not exhaustive, the websites shown in this table are a good cross-section of the various websites a user may visit, including mail, maps and wiki pages. Also, these websites are interesting based on their varied usage of JavaScript. Sites that do not use any JavaScript, such as `http://www.ottawahomerepairs.com` and `http://www.fitc.ca` are not profiled. We also observe from the table that JaSPIn takes longer to build a stable profile for websites which use more JavaScript. It is interesting to note, though, that the use of AJAX by itself does not increase the number of sequences in the profile of a web page.

Of particular interest is the profile of a website such as `http://www.cnn.com` which is a combination of profiles of related sub-domains and web pages for different stories in `http://www.cnn.com/2007/` . It can be seen that such a website requires more visits before a stable profile is created. On the other hand, `mail.yahoo.com` is an AJAX driven website whose profile is created by recording visits to a single URL `http://ca.mg1.mail.yahoo.com/dc/launch?.rand=5s28fnbhjmmdh`.This website takes only 11 visits before profile stability is achieved.

To understand profile stability over time, Figure 5.1 graphs the number of sequences in the profile of three different websites, Wikipedia (`http://en.wikipedia.com`), Yahoo Mail (`http://mail.yahoo.com`) and Rogers (`http://www.rogers.com`) over a period of 50 visits. The number of sequences in each of their profiles is plotted on the Y-axis against the visit number on the X-axis. Yahoo mail generates the least number of profiles (106) in spite of being an AJAX driven website. After the first couple of visits, the profiles stabilize at around 197 sequences for Yahoo Mail, 256

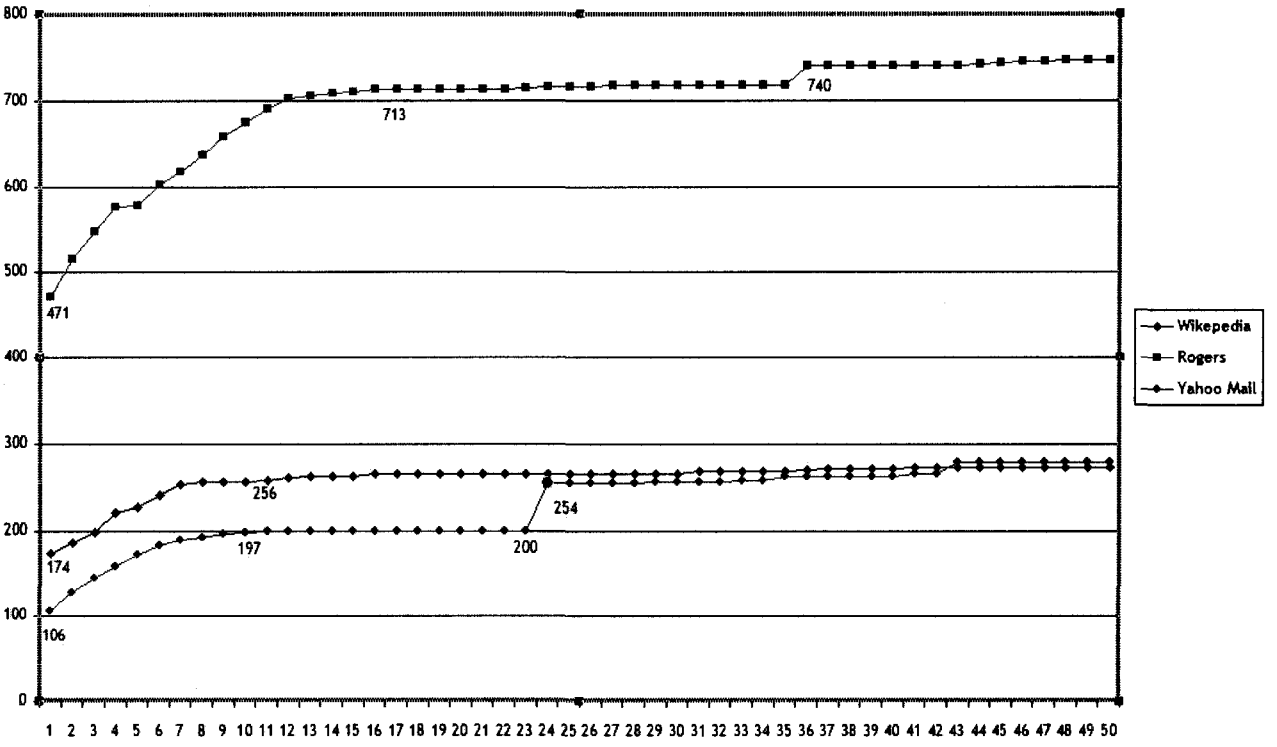

Figure 5.1: This graph shows the number of sequences observed per web site over a period of 50 visits. There are varying number of sequences in each web site's profile. The first visit generates a brand new policy, and the next few visits stabilize that policy.

sequences for Wikipedia and 713 for Rogers. Rogers takes the longest to stabilize and has the largest number of JavaScript sequences. It also has the largest variation in sequences from the initial visit to the 50th visit. The number of sequences in the profile for Yahoo Mail sees an increase at the 24th visit. This is due to the user creating a new folder, and moving messages to it. The previously unseen action generates new profiles and alters the normal for this user's interaction with Yahoo mail. False positives by changes in browsing patterns are discussed in detail in the next chapter.

5.4 False Positives

The traditional method for reporting IDS results is a receiver operating characteristic (ROC) curve that shows the trade off between identifying real attacks (true positives)

Type of website	Methods in map file	# of websites	Avg. False Positive Rate
Moderate JavaScript Use	<100	17	0.20
JavaScript Intensive	>100	31	0.214
AJAX	>100	12	0.091

Table 5.5: Average false positive rate across visits for websites with different levels of JavaScript usage. The false positiove rate is calculated by dividing the number of visits that generated a false alarm by the total number of visits to a web site after the profile is considered stable.

and incorrectly flagging non-attack requests as an attack (false positives) [HAN96]. However, in the case of JaSPIn, profiles are built for live websites hosted on the internet. During our evaluation period of over three months, none of the websites being profiled by JaSPIn were subject to a real attack. Plotting an ROC for dummy websites will not be useful as it may not reflect the true behavior of JavaScript used on real websites. Hence, we record the number of visits that generate a false positive, and use that as a measure to understand JaSPIn's accuracy at detecting attacks.

The number of false positives per visit were recorded from each of the sixty web sites that were profiled by JaSPIn. JaSPIn raises an alert during attack detection phase when it comes across a sequence (of length 6) not part of a web site's normal profile. False positives can be caused for two reasons: if the website updates its JavaScript method, or if the user triggers the call of a JavaScript method that was never called previously. In our evaluation, we consider all alerts from JaSPIn as false positives, even though the user is able to distinguish that as a false alarm and ask JaSPIn to ignore it.

Table 5.5 shows the average false positive rate across sites classified based on their JavaScript usage. We define the average false positive rate as the number of visits that generated a false positive alert as compared to the total number of visits. This usage is determined by the number of method names in the map file of a web site. 17 websites that use JavaScript moderately, 31 that make extensive use of JavaScript and 12 sites using AJAX were profiled and checked for sequence variations leading to a cross-site attack. The average false positive rate of 0.091 across sites using AJAX for a window size of 6, with its stability threshold set at 3, is promising, although higher than we would like.

Figure 5.2 shows the average false positive rate plotted against the maximum

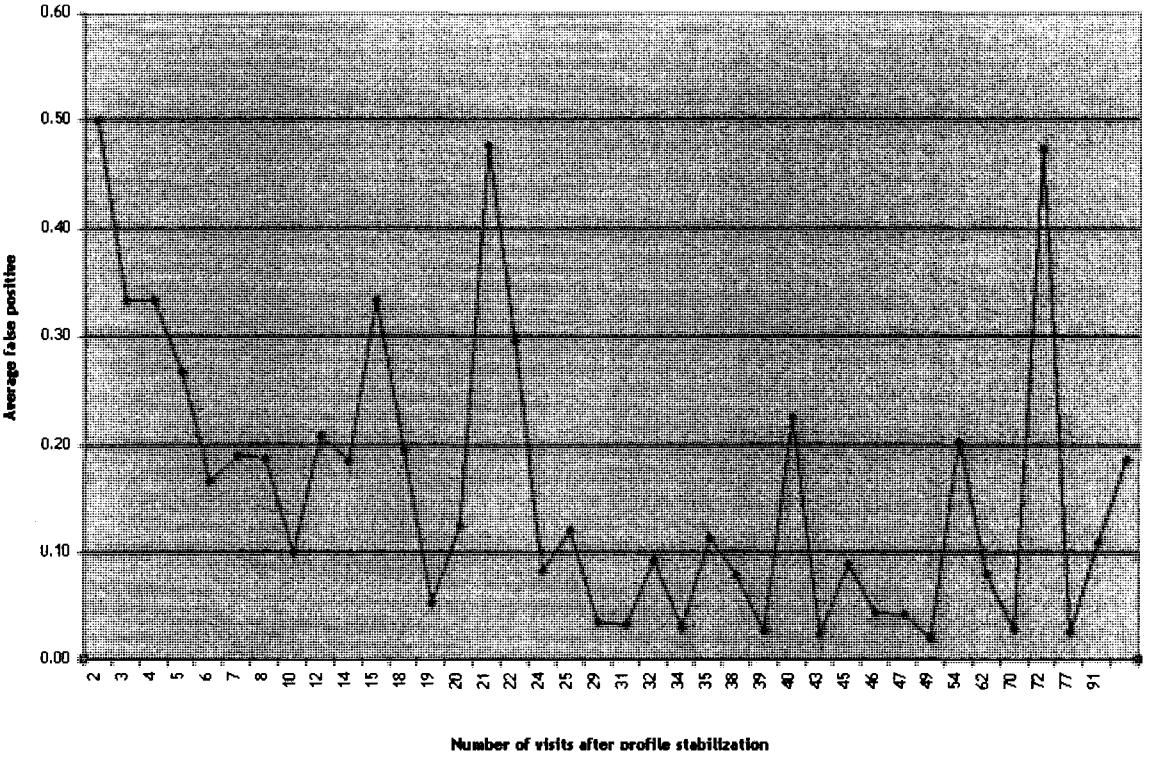

Figure 5.2: This graph shows the false positive rates across different sites plotted against the total number of visits to those sites in 3 months.

number of visits made to the site during the evaluation period. It can be seen that 50 percent of the sites visited had a false positive rate less than 0.1. We also observed that on average, the false positive rate for sites visited more number of times is lesser than those visited fewer times. This observation means that continued use of JaSPIn may lead to lower false positive rates. The relationship however is non-linear, since the false positive rate is dependent on not only the number of visits, but also the manner in which JavaScript is used on a page and consistency in user actions.

In general, the most common change to a website is its content, including text and images. A four year study by Koehler [Koe02] found that the half-life of a web page is approximately 2 years and web page content and functionality appears to stabilize over time; aging pages change less often than they once did. Less frequent updates to web pages help keep the false positive rate of JaSPIn low. Our experiments over the past year have shown that only a small percentage of sites (4 out of 60 monitored) undergo major changes to JavaScript code. We discuss this further in the

Website Type	Website	Map file size (KB)	Profile size (KB)
Dynamic	http://en.wikipedia.com	3	3
JS intensive	http://www.rogerstelevision.com	12	7
AJAX intensive	http://mail.yahoo.com	7	6
AJAX intensive	http://ajax.asp.net	6	8

Table 5.6: Space requirements for the Map file which has the list of all methods being invoked and the Profile file which has the sequences of length 6

next chapter.

False positives can also be caused by variations in the user's browsing pattern. Researchers at Carnegie Mellon University have shown that although internet usage has increased exponentially since the world wide web, browsing behaviors have remained surprisingly stable [MF01]. They have found that the number of pages and domains a user views during any particular session have remained stable. They also found that users remain stable to their hosts and persistent in their browsing habits. Although this research means greater accuracy can be expected from JaSPIn, user-caused false positives are perhaps unavoidable. User-caused false positives are harder to address and probably require more sophisticated heuristics. Such possibilities are examined in the discussion.

5.5 Overhead

JaSPIn is installed on the author's system, and the modified browser is used for browsing by the author on a daily basis. Daily usage shows no negative impact on regular browsing, with the exception of false positives (discussed above).We also wanted to see if JaSPIn affected the viewing experience of the user. The time required for a web page to load both during profile generation and profile verification is comparable to page loads with JaSPIn turned off. We installed the Load Time Analyzer [Tea06] on our modified Mozilla Firefox browser (v1.5) on a 512 MB RAM, 3400+ AMD Sempron Linux 2.6.11 computer. Load Time Analyzer is a Firefox extension created by Google that displays the number of events that are processed by a certain website and how long it takes to load them all. The load time displayed will obviously be influenced by the Internet connection, nonetheless it can provide a good indicator of the influence of JaSPIn on site speed. The home page of all 60 sites in the Appendix was loaded three times each both with JaSPIn turned off and JaSPIn in training

and attack detection modes. The total time to load the 60 pages three times each was 758 seconds, compared to 897.7 seconds to load the pages during training and 927.2 during attack detection. JaSPIn, roughly has an overhead of 0.8 seconds and 0.9 seconds when using JaSPIn in training and attack detection modes respectively. This slowdown is statistically more significant in pages that have more events being triggered (as indicated by the Load Time Analyzer) than in pages with a lesser number of events triggered before a page loads. `http://www.cnn.com` had 878 triggered events and the highest load time overhead of 4.1 seconds. Unlike rule based systems where the detection time is proportional to the size of the signature database, attack detection time in an anomaly detection system for a web page will not change significantly once a stable profile has been generated.

The sizes of profile and map files for some sample websites visited by the author on a regular basis are shown in Table 5.6. As is evident, given the current availability of storage space, the amount used by JaSPIn is small.

5.6 Profile diversity - Internet web surfing simulation

The purpose of this experiment is to determine if the profiles created by different browsing patterns are different. This is a significant measure to understand the effort required by an attacker to mimic normal behavior.

To simulate varied browsing patterns, we used an automatic web surfing application to generate profiles for websites. The crawler application opens a website in our modified JaSPIn enabled browser, collects all the links on a page, and randomly chooses the next link to open. This is continued for ten different clicks.

We chose to use `http:\\www.yahoo.com` as the website of choice for this experiment based on its popularity (which makes it a popular attack target as well) and extensive use of JavaScript in the pages.

Figure 5.3 shows the frequency of occurrence of each of the 449 sequences in the policy for http://www.yahoo.com during 25 visits. The goal of this experiment is to observe profile diversity, and understand if varying browsing behavior randomly causes varying profiles. Hence, we restricted the tool to browse only pages at

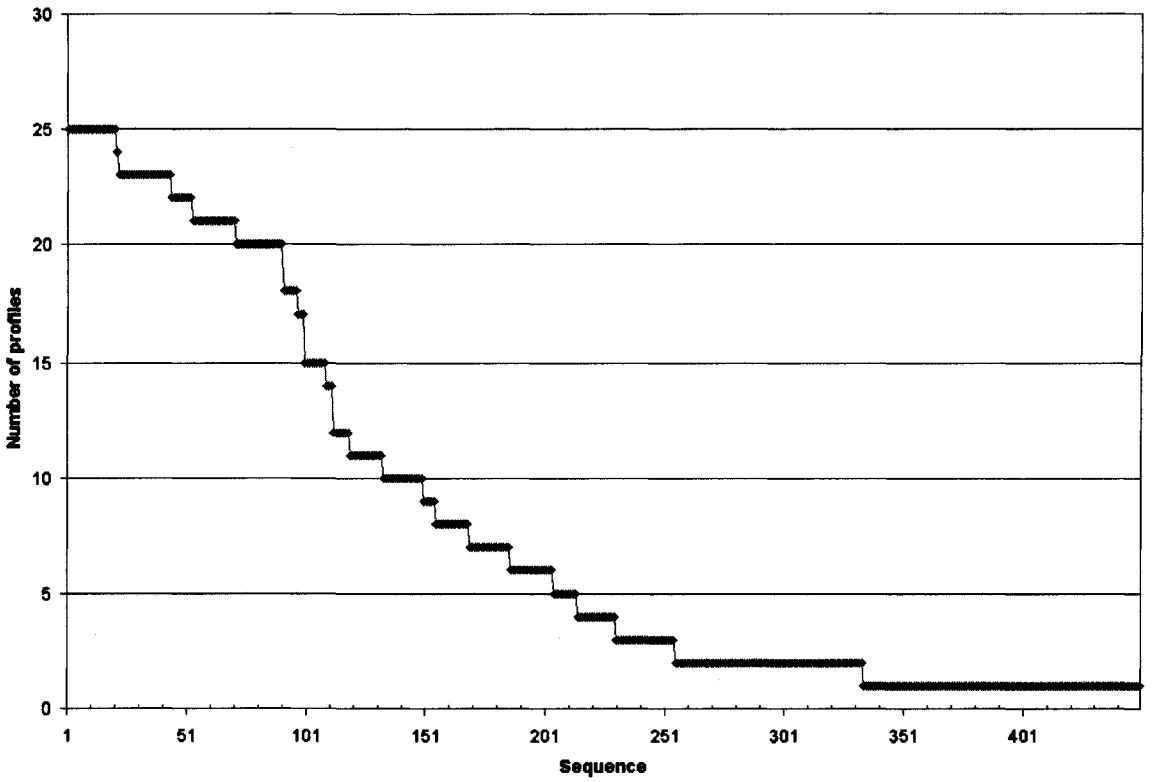

Figure 5.3: This graph shows the number of profiles containing each sequence of length 6 sorted by frequency. There are 25 profiles created from different visits to http://www.yahoo.com, which in total contain 449 unique sequences.

http://www.yahoo.com, since free browsing on the yahoo.com website increased profile diversity heavily. In fact, many new profiles were created for tech.yahoo.com, music.yahoo.com and video.yahoo.com, amongst others. Adding this restriction makes our model less natural since users do browse freely. However, it also makes profile diversity more difficult to achieve.

20 sequences (4.49%) appear in all the profiles, while 125 sequences (27.89%) appear only in a single profile. Each visit by the crawling aplication is different from the other due to the randomnness introduced. This variation in behavior introduced can be compared to variations in browsing patterns of different users. These results thus suggest that JaSPIn's definition of normal behavior varies from user to user. This diversity offers the possibility that a successful attack against one user may not work on another, even if both are accessing the same web page. We need to do more

work to determine if this diversity is true across all websites that use JavaScript, and whether this diversity adds much protection in practice.

5.7 Simulated Exploits

The most common XSS attacks were tested on a vulnerable website specially created for this purpose on our local server. The sample application is a simple community forum built in ASP (Active Server Pages) which accepts some data from the user, and posts it onto a web page.

Our test database of 59 XSS attacks is based on the vectors published by ha.ckers.org [RSn08]. Our set of tests included attacks successful against Firefox 1.5. Also among the 71 attacks available for testing, seven of the attacks are similar to the ones tested, and use same encoding techniques. Five other flash based attacks were not tested and are discussed below.

Table 5.8 summarizes the different categories of attacks that were tested. JaSPIn is able to successfully detect common XSS techniques such as case sensitivity, IP encoding, URL encoding, Dword encoding, hex encoding, octal encoding, mixed encoding, accent obfuscation, usage of null characters,usage of single quotes, double quotes, semicolon etc. Attackers can use various means to load malicious script onto a page, but actual damage happens only when the script executes on the user's browser. JaSPIn monitors these executions and is able to spot changes in method invocations caused by the attacker's script. We have found that most of the attacks are caught by our tracking of the relative order of property getters. Other malicious methods with the same name as those on a web page are also detected due to variations in the relative order in which they are invoked. JaSPIn's attack detection capabilities are discussed in the next chapter.

JaSPIn fails to detect any embedded flash attacks. Also, it is unable to detect the addition of links to a page to exploit the 'feeling lucky' feature that works if the exploitable page is the top of any keyword search. This attack is facilitated by a browser implementation flaw and hence is out of scope for JaSPIn as explained in our threat model in Chapter 3. JaSPIn is unable to detect a XSS attack directly embedded in ActionScript, the scripting language for Flash. This is because we have

not implemented our policy inference engine to handle ActionScript. However, given the similarity in syntax and behavior between ActionScript and JavaScript, detecting Flash XSS attacks does look promising using JaSPIn.

Type of Attack	Number of tests	Details
Basic XSS Attacks	5	No filter evasion. Caught by other detection techniques as well
Character Encoding Attacks	9	Obfuscation techniques with Unicode, Hex Encoding and UTF-7 Encoding
Embedded Character Attacks	14	Embedded encoded characters to break up the XSS attack
HTML Element Attacks	16	Injection using different input methods such as images, iframes, css, div etc.
Other Attacks	3	Cookie Manipulation, Renaming .js to .jpg, JavaScript Includes
URL Obfuscation	10	URL string evasion by hiding the URL of the site containing the malicious code
XSS w/HTML Quote Encapsulation	2	Filter evasion
Total Attacks Detected	**59**	

Table 5.8: Cross-site attacks detected by JaSPIn

5.8 Targeted attacks

We studied the effectiveness of our approach on the detection of previously seen XSS attacks on vulnerable applications. Based on their popularity and availability of known vulnerabilities, phpBB and WebCal have been selected for our experiments.

5.8.1 phpBB 2.0.19

phpBB [pG08] is the worlds leading open source discussion forum software. Many XSS vulnerabilities have been found on sites enabled by phpBB [Sec08]. phpBB version 2.0.19 was vulnerable to cross-site scripting attacks that relied on activated HTML messages that are enabled in the preferences for phpBB (see Figure 5.4).

We installed this software on our local server, and generated a stable policy for the site using JaSPIn with a window size of six and stability threshold of three. After 7 visits, a stable profile with 379 sequences was created. The exploit, as shown is Figure 5.5 is then posted as a message on the installed board. When the message is accessed with our modified browser and the mouse is moved over the text, JaSPIn

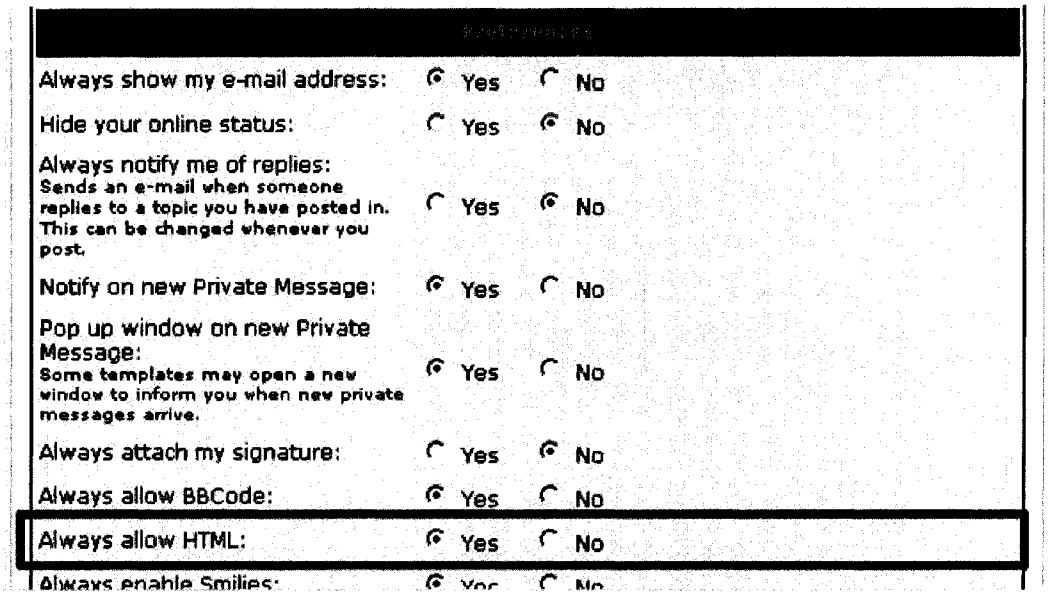

Figure 5.4: Preferences for the forum

```
<pre a='>'
onmouseover='document.location="http://evilhost/bad.php?c="+document.co
okie' b='
[img]http://locathost/test.jpg[/img]
```

Figure 5.5: Exploit code for phpBB

detects that a new sequence is detected onmouseover, and alerts the user as shown in Figure 5.6.

5.8.2 WebCal (v1.11-v3.04)

WebCal is a free browser based calendar program to track of appointments, meetings, birthdays etc. It uses perl based cgi scripts and some JavaScript to generate all the web pages. While researching this vulnerability, we found that one of the sites using WebCal was still vulnerable to this attack. Hence, this website was profiled, and a sample attack, Figure 5.7 tested against it.

JaSPIn is able to detect the anomaly due to a new method invocation (Figure 5.8

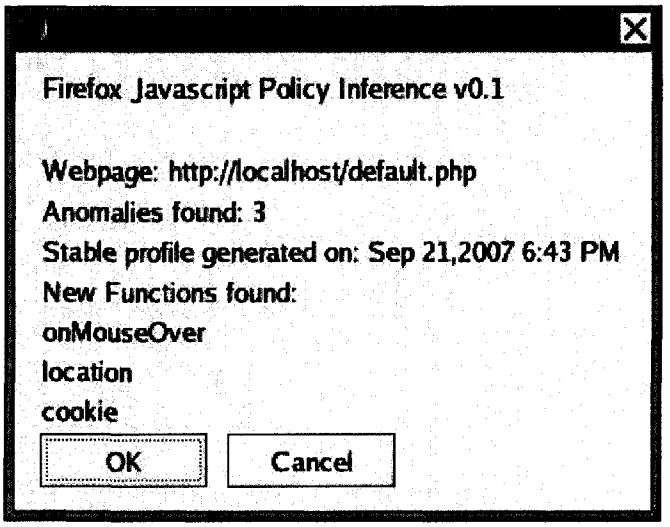

Figure 5.6: JaSPIn is able to detect the documented XSS exploit on phpBB

http://bulldog.tzo.org/perl/webcal.cgi?function=<script>alert(document.cookie)</script>

&cal=public

Figure 5.7: Sample attack against WebCal

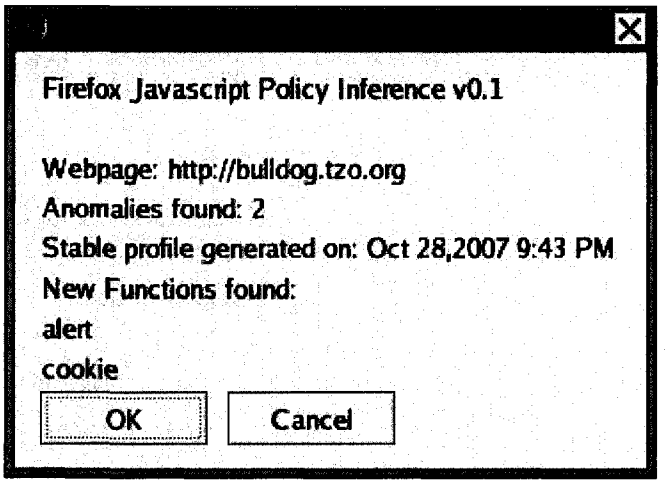

Figure 5.8: WebCal anomalies

5.8.3 Flash enabled JavaScript attacks

XSS attacks can be launched using Macromedia Flash [Mac08b]. Flash has its own built in scripting language, ActionScript [Mac08a] which has a getURL() action. This function allows the programmer to redirect the end user to another page. The

```
getURL("http://www.ccsl.carleton.ca")
```

Figure 5.9: Flash getURL function usage

```
getURL("javascript:alert(document.cookie)")
```

Figure 5.10: Using the getURL function to launch a XSS attack.

parameter would usually be a URL; something like `http://www.ccsl.carleton.ca`, so that the script looks like Figure 5.9.

This function can be used to launch an XSS attack by replacing the URL with a JavaScript function call as show in Figure 5.10

JaSPIn is able to detect such Flash based JavaScript attacks, since it is the actual execution of the JavaScript function call that we monitor and not the means to get to that point.

5.9 Summary

JaSPIn was tested on sixty different web sites for profile generation and monitoring. These profiles become more stable as the number of visits to a web page increase. JaSPIn was able to successfully detect fifty nine different common cross-site attacks. It was also successful when tested on phpBB, WebCal and Macromedia Flash enabled sites. These results suggest that JaSPIn can defend a monitored web page against many kinds of cross-site attacks by observing unusual patterns in JavaScript behavior.

Chapter 6

Discussion

The past two chapters documented the creation and testing of JaSPIn. This chapter reviews these results and places them in perspective. The first part of this chapter summarizes our observations from the experiments in Chapter 5. We then describe the concrete contributions of our research. Section 6.3 explains the limitations of the current prototype and suggests ways in which it could enhanced. The last section briefly looks at some potential future work.

6.1 Summary of results

Some of our observations include:

- Profiles for different pages on a web site may be different. JaSPIn is able to determine if sequences of method invocations for different pages on a web site needs to be maintained as a single profile or a set of profiles.

- The number of visits required to generate a stable profile for a web site depends on its usage of JavaScript and the individual user. Our experiments indicate that the number of newer sequences added to a profile decreases as the number of visits to the site increase.

- Profiles of the same web page generated by different users may be different. Our experiment on profile diversity used a random page selecting tool to simulate users behaving differently on web sites. We observed that profiles of the web site for different users had both common sequences and unique sequences.

- Window sizes affect profile generation if they are too low or too high, but do not affect profile generation prominently for values of six or nine.

- Cross site scripting attacks based on JavaScript are immediately detected, either because of a different JavaScript function being called or different sequence of the function.

Method invocations from Ad-servers are also profiled as part of a given web page. Ad-servers do not seem to have an impact on profile generation, however any changes to the JavaScript functions from an Ad-server called in the context of a given web page will affect the profile. The JavaScript code executed from the Ad-server is treated as part of the web site it is embedded into,and added to the policy of that website. Although it is possible for us to determine the source of the JavaScript file (the URL of the .js file), we are not using this information explicitly. Perhaps future work could make use of this information to analyze the same set of sequences from Ad-servers being a part of various websites.

6.2 Contributions

The research presented in this work makes several contributions to the domain of web security, specifically in the area of detecting cross-site attacks. The previous chapter has presented evidence that short sequences of JavaScript methods are good discriminators between normal and abnormal behavior of several common web pages visited by a user. In essence, we have found a regularity in the order in which method invocations happen in the JavaScript interpreter that is highly likely to be perturbed by intrusive activities. Chapter 5 has shown that monitoring of JavaScript code execution can be performed efficiently in real-time and can detect JavaScript based cross-site attacks.

Our experiments were carried out on a range of sites varying in purpose and use of JavaScript to provide a good sample test bed for JaSPIn that is representative of most sites a user may visit. Our results are suggestive to the use of anomaly detection to detect web based attacks. The rest of this section discusses past chapters and explains these contributions in more detail.

As explained in Section 2.5, use of server side solutions to deal with XSS attacks leave the user completely vulnerable if the website has not implemented any such measures or is not able to fix the security issues on time. Our approach is implemented

at the client and requires no effort from the programmers to modify online web applications.

Section 5.3 presents one of the principal contributions of this thesis, showing that stable profiles for websites can be created automatically. The number of new sequences that are added to the profile decreases as the number of visits increase. The false positive rate is as low as zero for sites that have a simple interface and regular use of JavaScript methods. Server side static analysis cross-site detection techniques have reported false positive rates as high as 50 percent. Also, Noxes reports a false positive rate of 5.7 percent but is unable to detect reflected XSS attacks. Other approaches, such as dynamic taint analysis [VNJ+07] and BEEP [JSH07] have not presented their false positive rates explicitly. Hence, a direct comparison is not possible. However, our system has better detection capabilities than both of these approaches. Dynamic taint analysis has not been tested against AJAX based attacks. Also, our profiles are tighter and user specific as compared to BEEP, thereby making circumvention of the IDS using mimicry techniques more difficult.

It is difficult to measure false alarms accurately because an IDS may have a different false positive rate for each web page and there is no such thing as a standard web page. Also, it is difficult to determine aspects of user activity that will cause false alarms. As a result, it may be difficult to guarantee that we can produce the same number and type of false alarms as found in our tests. However, JaSPIn can be configured and tuned in a variety of ways in order to reduce the false positive rate. Our experiments suggest that a JaSPIn enabled browser with a window size of six and a threshold of three can be used for day to day browsing with a relatively low rate of false positives as shown in Section 5.4. Unlike previous client based approaches, the user does not need to maintain any white lists.

A significant benefit of our anomaly-based approach is the ability to detect never before seen cross-site scripting attacks. Also, as seen in Section 5.7, JaSPIn is able to detect a wide variety of XSS attacks. Our test suite is based on the XSS attack vectors published by ha.ckers.org [RSn08]. This attack database incorporates different types of XSS attacks which use obfuscation to evade common server-side filters and novel mechanisms of infiltration and propagation. Except for cross-site attacks embedded

in a flash movie, and attacks targeted at the browser itself, JaSPIn is able to detect all other attacks in the database.

Lastly, before any new security mechanism can be widely deployed, it must be shown to have minimal performance impact. This is especially true in the web security domain as any tool that makes the user's browsing experience slower will not be readily accepted. JaSPIn's overhead is small enough that normal users do not notice any difference in system performance. The results in Section 5.5 show that both profile generation and attack detection using method invocation monitoring can be performed in real-time with little overhead: web pages profiled in our experiments are slowed down by less than 1 second. JaSPIn's performance is competitive with other client side extensions to detect cross-site attacks such as BEEP [JSH07] and NoMoXSS [VNJ+07].

In conclusion, these results illustrate the viability of monitoring JavaScript code execution as a mechanism to detect cross-site scripting attacks.

6.3 Limitations

The following sections discuss limitations of both our approach in general, and the current version of JaSPIn in particular and how they could be overcome.

6.3.1 Limitations of the approach

Profiling Normal

A limitation of anomaly detection systems is that they are unable to detect attacks until a normal profile can be established. This means that if a web page that does not have a stable profile is subject to a cross-site attack, JaSPIn will not be able to detect it. One way to protect more web sites would be to allow JaSPIn to detect anomalies while new sequences were still being added to a profile. A weakness of this strategy is that the number of false positives will also increase significantly. Perhaps the simplest way to ensure that JaSPIn has profiles of normal program behavior would be for web developers to distribute default profiles of normal program behavior, as required by BEEP [JSH07]. These synthetic normal profiles could be generated similar to that

required for BEEP. These profiles lack information on the order in which method calls are invoked, but over a few visits, JaSPIn will replace many of these profiles with ones that are specialized to the usage patterns of the user. These profiles would generally be smaller than the default synthetic normal profiles and would restrict JavaScript behavior better.

Attacker Circumvention

Given that it is a client-level system, the attacker can attack the policy generator itself. Since the user has no way of validating if the messages are from JaSPIn or a compromised version of JaSPIn, the attacker can fool the user with false messages. However, this requires sophisticated techniques and is not as easy as an XSS attack.

Like many anomaly detection systems that work at the program level, JaSPIn is subject to mimicry attacks [WS02]. Mimicry attacks can be defined as attacks that achieve attacker-intended effects without modifying aspects of an application behavior that are monitored by an IDS. The principal challenge is that of making them practical: for JavaScript inferred policies this is surely a possibility. Although JaSPIn has been designed to monitor JavaScript calls at the interpreter level, which includes calls from the DOM that result as part of an external script, and is dependent on user browsing patterns, it is possible for the attacker to mimic normal behavior by calling methods in the same sequence as the most common paths for all users. For example, if a web page uses the onload function, it is bound to execute for all users and if the attacker can get the set of methods following an onload, he can modify his attack code to generate the same sequence. Increasing the window size will reduce the chances of a successful mimicry attack, but it also means that we will require large amounts of training and run the risk of increasing the number of false positives significantly. Profile diversity, explained in Section 5.6 increases the effort required to launch a mimicry attack. Some other techniques to address mimicry attacks are explained in the Future Work section.

User involvement

A big plus of JaSPIn is that policies are automatically generated. However, when abnormal behavior is detected, user involvement is required to determine if the anomaly is a real attack or a profile update. Ideally, we would like JaSPIn to judge for itself whether the anomaly is an attack. An anomaly detection system by itself cannot make this distinction. In order to be able to automatically judge whether an anomaly is an attack, JaSPIn can be integrated as part of a layered defense mechanism where useful information from other sources could help JaSPIn make an informed decision.

6.3.2 Limitation of our current implementation

Browser Limitation

Our current method has been implemented in Firefox and requires the average Firefox user to install our modified version till such time that our system is robust enough to be integrated into the main release. This also means that for every new version of Firefox, we need to provide a new version of JaSPIn enabled Firefox. The advantage of such a browser solution, however, is that no code changes are required on the part of web developers.

Mass Evaluation

JaSPIn has not been evaluated across a large user population. Although this will not affect its attack detection capabilities, it may add more information on our false positive rate. Our current results are sufficient, however, to illustrate the applicability of anomaly-based monitoring to cross-site attack detection.

Usability

The Firefox extension that presents the information about profiles generated and anomalies detected is only a prototype implementation. It could be improved to present more specific information to the user in a way that would also catch the user's attention immediately. Finally, a feature to view the history of changes to the profile of websites could improve user knowledge.

Lack of a response mechanism

While the current IDS implementation can detect when an attack is occurring and identify the offending function call, it currently has no offensive mechanism to stop the program's behavior. Implementing a reaction mechanism would allow the IDS to reduce the impact of an attack.

6.4 Future Work

6.4.1 Resistance to mimicry attacks

In the future we intend to look at ways to make JaSPIn resistant to mimicry attacks. Use of additional features of program execution, such as function arguments and frequency of use increase the effort required by an attacker to mimic normal behavior. Introducing automatic randomization of the window size would make the profiles more difficult to mimic.

6.4.2 Reduction of the false positive rate

False positives can be reduced by increasing learning. This can be achieved by including a time factor into the training period. Thus, training will be longer and web page changes can be tracked. pH [Som02] was able to achieve a significant reduction the its false positive rate by using such techniques. This is encouraging and something we would definitely like to try in the future.

6.4.3 Detection of other JavaScript attacks

In Chapter 2, we introduced CSRF attacks 2.3.4 and other JavaScript based attacks 2.2 that do not violate the security policy. This section evaluates the possibility of using JaSPIn to detect such attacks.

JavaScript attacks that make use of the language's features, such as displaying pop up windows for every mouse click or displaying multiple alert messages to the user are annoying. JaSPIn can determine how many times a particular method call is being invoked. Although the current version of JaSPIn does not have any way for the user to specify behavioral policies such as those that restrict the opening of multiple

windows, it could be integrated into the system in the future. The core JavaScript monitoring engine does have the knowledge required to detect such attacks and hence the lacking functionality can be added as an additional layer in the future.

CSRF attacks are more tricky to detect than XSS attacks because the web site is sent a malicious request from a trusted user. JaSPIn creates profiles of websites based on user behavior. If the behavior of the malicious code is similar to that of the user, CSRF attacks go undetected. However, if the malicious code performs an action untypical of the user, JaSPIn detects the same. Our approach is not designed to detect CSRF attacks and more research needs to be done to study the applicability of anomaly based approaches to CSRF attack detection.

6.4.4 Usability

We would like to make JaSPIn user friendly by adding ways for the user to understand the web site profiles and detected variations from the same. One way to improve this understanding would be to provide information on which user action caused a variation from the normal profile.

We would like to make JaSPIn available for download to get some feedback on its accuracy and usability in day to day browsing for different users.

Also, we found JaSPIn to be a good tool just to understand the usage of JavaScript on the internet today. Our logs provide valuable information on methods being invoked by a page and their relative order. This information could be analyzed by web site owners to determine which features are most popular among its users and understand usability of their website in general.

Chapter 7

Conclusion

Cross-site scripting vulnerabilities on the web are being discovered and disclosed every day. While XSS attacks by themselves have been long recognized in the web application security space, there is no indication that the problem is getting better. Application layer attacks are difficult to detect and protect against using traditional security mechanisms. Cross-site attacks are generally simple, but difficult to detect because of the high flexibility that HTML encoding schemes provide to the attacker for circumventing server-side input filters.

In this thesis we have presented a novel mechanism to infer JavaScript policies to protect against cross-site scripting attacks. We used the principles of anomaly based intrusion detection systems to profile web sites based on their sequence of JavaScript calls. We extended the JavaScript engine in Firefox (SpiderMonkey) and implemented a Firefox extension to manage the profiles. The browsing speed overhead of using our tool is minimal and observed false alarm rate low.

Although more work needs to be done in minimizing user intervention on detection of policy violations, we consider this work to be a good first step.

Appendix A

Appendix A - Common JavaScript Methods

Table A.1: Summary of common objects and their respective methods in JavaScript. This table has been derived from the chapters in [Fla98]

Object	**Properties**	**Methods**	**Event Handlers**
Window	defaultStatus	alert	onLoad
	frames	blur	onUnload
	opener	close	onBlur
	parent	confirm	onFocus
	scroll	focus	
	self	open	
	status	prompt	
	top	clearTimeout	
	window	setTimeout	
Frame	defaultStatus	alert	none
	frames	blur	
	opener	close	
	parent	confirm	
	scroll	focus	
	self	open	
	status	prompt	
	top	clearTimeout	
	window	setTimeout	
Location	hash	reload	none

Continued on Next Page...

Table A.1 – Continued

Object	**Properties**	**Methods**	**Event Handlers**
	host hostname href pathname por protocol search	replace	
History	length forward go	back	none
Navigator	appCodeName appName appVersion mimeTypes plugins userAgent	javaEnabled	none
document	alinkColor anchors applets area bgColor cookie fgColor forms images lastModified linkColor	clear close open write writeln	none

Continued on Next Page...

Table A.1 – Continued

Object	Properties	Methods	Event Handlers
	links location referrer title vlinkColor		
image	border complete height hspace lowsrc name src vspace width	none	none
form	action elements encoding FileUpload method name target	submit reset	onSubmit onReset
text	defaultValue name type value	focus blur select	onBlur onCharge onFocus onSelect
Built-in Objects			

Continued on Next Page...

Table A.1 – Continued

Object	**Properties**	**Methods**	**Event Handlers**
Array	length	join	none
		reverse	
		sort xx	
Date	none	getDate	none
		getDay	
		getHours	
		getMinutes	
		getMonth	
		getSeconds	
		getTime	
		getTimeZoneoffset	
		getYear	
		parse	
		prototype	
		setDate	
		setHours	
		setMinutes	
		setMonth	
		setSeconds	
		setTime	
		setYear	
		toGMTString	
		toLocaleString	
		UTC	
String	length	anchor	
	prototype	big	
		blink	

Continued on Next Page...

Table A.1 – Continued

Object	Properties	Methods	Event Handlers
		bold	
		charAt	
		fixed	
		fontColor	
		fontSize	
		indexOf	
		italics	
		lastIndexOf	
		link	
		small	
		split	
		strike	
		sub	
		substring	
		sup	
		toLowerCase	
		toUpperCase	

Appendix B

Profile and Map file Sample

B.1 shows a sample Map file. The map file contains the list of all methods that have been invoked during the execution of the web page. B.2 shows the profile file with a window size of 6. Each of the numbers in the profile file maps to a method in the map file for that web site.

```
_EM
_FF
_FU
_FO
_FA
_GR
_PO
_PP
_E
_F
_G
_H
_I
_J
_Q
_U
_X
getElementsByClass
changeTab
document
```

Figure B.1: Part of a map file

```
31 30 29 28 27 26
33 32 31 30 29 28
35 34 33 32 31 30
41 40 39 38 37 36
43 42 41 40 39 38
45 44 43 42 41 40
47 46 45 44 43 42
49 48 47 46 45 44
51 50 49 48 47 46
53 52 51 50 49 48
55 54 53 52 51 50
57 56 55 54 53 52
3 3 3 37 3 3
3 3 3 3 37 3
3 3 3 3 3 37
```

Figure B.2: Part of a sample profile file

Appendix C

List of web sites used in our evaluations of JaSPIn

Table C.1: List of web sites profiled in our Evaluation of JaSPIn. The first column gives the top level URL of the web site, the second column categorizes based on its use of JavaScript (JS), the third column indicates the number of visits to that site after a stable profile has been established and the fourth column gives the false positive rate.

Web site profiled	Type	Visits	FP
http://charanshappyhours.org	JS Intensive	3	0.000
http://mail.yahoo.com	AJAX	49	0.020
http://opl.ottawa.on.ca	JS Intensive	46	0.022
http://www.petplace.com/	JS Intensive	43	0.023
http://balasubramanians.com/	Moderate JS Use	39	0.026
https://dev.rwdg.net	JS Intensive	77	0.026
http://www.yahoo.com	AJAX	70	0.029
http://timesofindia.indiatimes.com/?	Moderate JS Use	34	0.029
http://www.cs.unibo.it/ds-rt2007/	Moderate JS Use	31	0.032
http://nemoboy.blogspot.com	Moderate JS Use	29	0.034
http://www.orkut.com	AJAX	47	0.043
https://webmail.magma.ca	AJAX	22	0.045
http://www.raaga.com	Moderate JS Use	20	0.050
http://ajaxwhois.com/	AJAX	19	0.053
http://www.carlinganimalhospital.com/	JS Intensive	35	0.057
http://www.youtube.com	JS Intensive	32	0.063

Continued on Next Page...

Table C.1 – Continued

Web site profiled	Type	Visits	FP
http://www.cardelhomes.com	Moderate JS Use	46	0.065
http://ccsl.carleton.ca	Moderate JS Use	15	0.067
http://www.eidosglobal.com	Moderate JS Use	15	0.067
http://www.google.com	JS Intensive	14	0.071
http://www.costco.ca	JS Intensive	38	0.079
http://www.gmail.com	AJAX	62	0.081
http://maps.google.com/	AJAX	12	0.083
http://slashdot.org	JS Intensive	24	0.083
http://www.theweathernetwork.com/	Moderate JS Use	45	0.089
http://www.ieee.com	JS Intensive	10	0.100
https://www.webupdates.tv	JS Intensive	91	0.110
http://sears.ca	JS Intensive	25	0.120
http://safari.oreilly.com/	JS Intensive	8	0.125
http://lxer.com/	JS Intensive	32	0.125
http://www.flickr.com	AJAX	7	0.143
http://www.microsoft.com	AJAX	7	0.143
http://www.books24x7.com/	JS Intensive	14	0.143
http://www.gerber.com	JS Intensive	14	0.143
http://ajax.asp.net	AJAX	6	0.167
http://ca.mcafee.com	JS Intensive	6	0.167
http://ieee.ca	JS Intensive	18	0.167
http://tdcanadatrust.com	JS Intensive	35	0.171
			0.186
http://www.cnn.com	JS Intensive	5	0.200
http://www.cs.unibo.it/projects/mswim2007/	Moderate JS Use	5	0.200
http://www.pizzapizza.com	JS Intensive	20	0.200
http://www.rogerstv.com	JS Intensive	54	0.204

Continued on Next Page...

Table C.1 – Continued

Web site profiled	Type	Visits	FP
http://www.smc2007.org/	Moderate JS Use	18	0.222
http://www.library.carleton.ca/	JS Intensive	40	0.225
http://www.rediff.com	JS Intensive	4	0.250
http://www.cra-arc.gc.ca	JS Intensive	4	0.250
http://www.gerber.com	JS Intensive	8	0.250
http://www.petsymposium.org/2007/	Moderate JS Use	7	0.286
http://www.facebook.com/	AJAX	14	0.286
http://www.imdb.com	Moderate JS Use	14	0.286
http://en.wikipedia.com	Moderate JS Use	12	0.333
http://www.petsmart.com	JS Intensive	5	0.400
http://www.rogers.com	JS Intensive	72	0.472
http://www.achannel.ca	JS Intensive	21	0.476
http://www.indianrail.gov.in/	JS Intensive	2	0.500
http://www.ieee.ca/epc07	Moderate JS Use	2	0.500
http://www.cpac.ca	Moderate JS Use	4	0.500
http://www.goodlifefitness.ca/	JS Intensive	22	0.545
http://www.site.uottawa.ca/~cadams/sac2007/	Moderate JS Use	3	0.667
http://www.tamilmatrimony.com/	JS Intensive	15	0.867

Appendix D

List of attacks used in our evaluations of JaSPIn

Table D.1: List of attacks tested in our Evaluation of JaSPIn. The first column gives the attack type, and the second column the attack name to identify what it does. Detailed code for each of these attacks can be found at [RSn08]

Attack Type	**Attack name**
Basic XSS Attacks	XSS Locator
Basic XSS Attacks	XSS Quick Test
Basic XSS Attacks	SCRIPT w/Alert()
Basic XSS Attacks	SCRIPT w/Source File
Basic XSS Attacks	SCRIPT w/Char Code
Character Encoding Attacks	Case Insensitive
Character Encoding Attacks	HTML Entities
Character Encoding Attacks	Grave Accents
Character Encoding Attacks	Image w/CharCode
Character Encoding Attacks	DIV w/Unicode
Character Encoding Attacks	Hex Encoding w/out Semicolons
Character Encoding Attacks	Escaping JavaScript escapes
Character Encoding Attacks	End title tag
Character Encoding Attacks	STYLE w/broken up JavaScript
Embedded Character Attacks	Embedded Tab
Embedded Character Attacks	Embedded Encoded Tab
Embedded Character Attacks	Embedded Newline
Embedded Character Attacks	Embedded Carriage Return

Continued on Next Page...

Table D.1 – Continued

Attack Type	**Attack name**
Embedded Character Attacks	Multiline w/Carriage Returns
Embedded Character Attacks	Null Chars 1
Embedded Character Attacks	Null Chars 2
Embedded Character Attacks	Spaces/Meta Chars
Embedded Character Attacks	Non-Alpha/Non-Digit
Embedded Character Attacks	Non-Alpha/Non-Digit Part 2
Embedded Character Attacks	No Closing Script Tag
Embedded Character Attacks	Extraneous Open Brackets
Embedded Character Attacks	Malformed IMG Tags
Embedded Character Attacks	No Quotes/Semicolons
HTML Element Attacks	BGSOUND
HTML Element Attacks	BODY background-image
HTML Element Attacks	BODY ONLOAD
HTML Element Attacks	DIV background-image 1
HTML Element Attacks	DIV expression
HTML Element Attacks	FRAME
HTML Element Attacks	IFRAME
HTML Element Attacks	INPUT Image
HTML Element Attacks	US-ASCII encoding
HTML Element Attacks	META
HTML Element Attacks	OBJECT
HTML Element Attacks	STYLE
HTML Element Attacks	Stylesheet
HTML Element Attacks	Remote Stylesheet 1
HTML Element Attacks	TABLE
HTML Element Attacks	TD
Other Attacks	Cookie Manipulation

Continued on Next Page...

Table D.1 – Continued

Attack Type	**Attack name**
Other Attacks	Rename .js to .jpg
Other Attacks	JavaScript Includes
URL Obfuscation	IP Encoding
URL Obfuscation	URL Encoding
URL Obfuscation	Dword Encoding
URL Obfuscation	Hex Encoding
URL Obfuscation	Octal Encoding
URL Obfuscation	Mixed Encoding
URL Obfuscation	Protocol Resolution Bypass
URL Obfuscation	Firefox Lookups 1
URL Obfuscation	Firefox Lookups 2
URL Obfuscation	Removing Cnames
XSS w/HTML Quote Encapsulation	Evade Regex Filter
XSS w/HTML Quote Encapsulation	Filter Evasion

Bibliography

[Ado07] Adobe. Cross-site scripting vulnerability in versions 7.0.8 and earlier of adobe reader and acrobat, 2007. `http://www.adobe.com/support/security/advisories/apsa07-01.html`.

[Axe00] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers Univ., March 2000.

[BAP+03] Elena Gabriela Barrantes, David H. Ackley, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *10th ACM conference on Computer and communications security*, pages 281–289. ACM Press., October 27-30 2003.

[CER00] Cert advisory ca-2000-02 malicious html tags embedded in client web requests., February 2000.

[CER06] US CERT. Vulnerability note vu808921, March 2006. `http://www.kb.cert.org/vuls/id/808921`.

[Cor06] Microsoft Corporation. Mitigating cross-site scripting with http-only cookies, 2006.

[Cor08] Mozilla Corporation. Spidermonkey, 2008. `http://www.mozilla.org/js/spidermonkey/`.

[DTH06] Rachna Dhamija, J. D. Tygar, and Marti Hearst. Why phishing works. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 581–590, New York, NY, USA, 2006. ACM.

[ECM99] ECMA. Standard ecma-262: Ecmascript language specification, 1999. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[FHSL96] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for unix processes. In *The 1996 IEEE Symposium on Security and Privacy*, page 120. IEEE Computer Society, May 6-8 1996.

[Fla98] David Flanagan. *JavaScript: The Definitive Guide.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

[Fog07] Seth Fogie. *Cross Site Scripting Attacks.* Syngress, City, 2007.

[Fou08] Mozilla Foundation. Firefox, 2008. `http://www.mozilla.com/en-US/firefox/`.

[FP07] K. Fernandez and D. Pagkalos. Xssed project, 2007. `http://www.xssed.com`.

[HAN96] J HANCOCK. *Signal Detection Theory.* McGraw-Hill, 1996.

[Har88] Norm Hardy. The confused deputy. Technical report, Key Logic, Inc., 1988.

[HFS98] Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. *J.Comput.Secur.*, 6(3):151–180, 1998.

[HV05] Oystein Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 85–94, Washington, DC, USA, 2005. IEEE Computer Society.

[HYH+04] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[IEKY04] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A proposal and implementation of automatic detection/collection system for cross-site scripting vulnerability. In *AINA '04: Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, page 145, Washington, DC, USA, 2004. IEEE Computer Society.

[IF02] Hajime Inoue and Stephanie Forrest. Anomaly intrusion detection in dynamic execution environments. In *NSPW '02: Proceedings of the 2002 workshop on New security paradigms*, pages 52–60, New York, NY, USA, 2002. ACM.

[Inc02] Sanctum Inc.(IBM). Appshield - application security firewall, 2002. `http://whitepapers.silicon.com/0,39024759,60044497p,00.htm`.

[Int99] Ecma International. Ecmascript language specification, 1999.

[JB07] Martin Johns and Christian Beyerlein. Smask: preventing injection attacks in web applications by approximating automatic data/code separation. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 284–291, New York, NY, USA, 2007. ACM.

[JKK06] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27–36, New York, NY, USA, 2006. ACM.

[JSH07] Trevor Jim, Nikhil Swamy, and Michael Hicks. Defeating script injection attacks with browser-enforced embedded policies. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 601–610, New York, NY, USA, 2007. ACM.

[KKP03] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, New York, NY, USA, 2003. ACM.

[KKVJ06] Engin Kirda, Christopher Kruegel, Giovanni Vigna, and Nenad Jovanovic. Noxes: a client-side solution for mitigating cross-site scripting attacks. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 330–337, New York, NY, USA, 2006. ACM.

[KMVV03] Christopher Kruegel, Darren Mutz, Fredrik Valeur, and Giovanni Vigna. On the detection of anomalous system call arguments. In *8th European Symposium on Research in Computer Security (ESORICS)*, 2003.

[Koe02] Wallace Koehler. Web page change and persistence-a four-year longitudinal study. In *Journal of the American Society for Information Science and Technology*, pages 162–171. John Wiley and Sons, Inc., 2002.

[Kre] Brian Krebs. Washington post security blog.

[KV03] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *10th ACM Conference on Computer and Communications Security (CCS)*, 2003.

[KVV04] Christopher Kruegel, Fredrik Valeur, and Giovanni Vigna. *Intrusion Detection and Correlation: Challenges and Solutions (Advances in Information Security)*, volume 1. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[Law07] George Lawton. Web 2.0 creates security challenges. *Computer*, 40(10):13–16, 2007.

[Mac08a] Macromedia. Actionscript technology center, 2008. `http://www.adobe.com/devnet/actionscript/`.

[Mac08b] Macromedia. Adobe flash player, 2008. `http://www.macromedia.com/software/flash/about/`.

[Mao08] Giorgio Maone. Noscript - firefox extension, 2008. `http://noscript.net/`.

[MF01] Alan L. Montgomery and Christos Faloutsos. Identifying web browsing trends and patterns. *Computer*, 34(7):94–95, 2001.

[Min05] Yasuhiko Minamide. Static approximation of dynamically generated web pages. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 432–441, New York, NY, USA, 2005. ACM.

[Net08] NetCraft. June 2008 web server survey, 2008.

[Nom06] Nomensa. United nations global audit of web accessibility, Dec 06 2006.

[OSW08] T. Oda, A. Somayaji, and T. White. ¡b style="color:black;background-color:ffff66"¿content provider conflict on the modern web¡/b¿. In *3rd Annual Symposium on Information Assurance (ASIA'08)*, Jun 2008.

[PFH03] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[pG08] phpBB Group. phpbb.com - creating communities., 2008. `http://www.phpbb.com`.

[Pro03] Niels Provos. Improving host security with system call policies. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2003. USENIX Association.

[RDW+07] C. Reis, J. Dunagan, H. Wang, O. Dubrovsky, and S. Esmeir. Browsershield: Vulnerability-driven filtering of dynamic html. *ACM Transactions on the Web*, 1(3), 2007.

[RSn08] RSnake. Xss (cross site scripting) cheat sheet. esp:for filter evasion., 2008. `http://ha.ckers.org/xss.html`.

[Rud01] Jesse Ruderman. The same origin policy, August 24 2001. `http://www.mozilla.org/projects/security/components/same-origin.html`.

[RVKK06] William Robertson, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *In Proceedings of the 13 th Symposium on Network and Distributed System Security (NDSS*, 2006.

[Sec08] Secunia. Vulnerabilities summary for phpbb2.x, 2008. `http://secunia.com/product/463/?task=statistics`.

[SMS99] Matthew Stillerman, Carla Marceau, and Maureen Stillman. Intrusion detection for distributed applications. *Communications of the ACM*, 42(7):62–69, 1999.

[Som02] Anil Buntwal Somayaji. Operating system stability and security through process homeostasis, 2002.

[SS02a] David Scott and Richard Sharp. Abstracting application-level web security. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 396–407, New York, NY, USA, 2002. ACM.

[SS02b] David Scott and Richard Sharp. Spectre: A tool for inferring, specifying, and enforcing web-security. Technical report, Cambridge University, 2002.

[SW06] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM.

[Tea06] Google Team. Load time analyzer, 2006. `https://addons.mozilla.org/en-US/firefox/addon/3371`.

[UELX07] Úlfar Erlingsson, Benjamin Livshits, and Yinglian Xie. End-to-end web application security. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[VNJ+07] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross-site scripting prevention with dynamic data tainting and static analysis. In *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2007.

[W3C04] W3C. Document object model, 2004. `http://www.w3.org/DOM/`.

[W3C07] W3C. The xmlhttprequest object, 2007. `http://www.w3.org/TR/XMLHttpRequest/`.

[Wag99] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical report, University of California at Berkeley, 1999. source: http://www.ncstrl.org:8900/ncstrl/servlet/search?formname=detail&id=oaicstrlh

[War01] Peter Warren. Teaching programming using scripting languages. *J. Comput. Small Coll.*, 17(2):205–216, 2001.

[WD01] David Wagner and Drew Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[WFP99] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *In IEEE Symposium on Security and Privacy*, pages 133–145. IEEE Computer Society, 1999.

[WS02] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 255–264, New York, NY, USA, 2002. ACM.

[WS08] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 171–180, New York, NY, USA, 2008. ACM.

[XBS06] Wei Xu, Sandeep Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: a practical approach to defeat a wide range of attacks. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[YCIS07] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–249, New York, NY, USA, 2007. ACM.