

San Jose State University
SJSU ScholarWorks

Master's Projects

Master's Theses and Graduate Research

Fall 12-17-2014

Spartan Web Application Firewall

Brian C. Lee

San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects

Part of the [Information Security Commons](#)

Recommended Citation

Lee, Brian C., "Spartan Web Application Firewall" (2014). *Master's Projects*. 376.

DOI: <https://doi.org/10.31979/etd.ucxt-jvp6>

https://scholarworks.sjsu.edu/etd_projects/376

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Spartan Web Application Firewall

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Brian C. Lee

December 2014

© 2014

Brian C. Lee

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Spartan Web Application Firewall

by

Brian C. Lee

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2014

Thomas Austin Department of Computer Science

Mark Stamp Department of Computer Science

Chris Tseng Department of Computer Science

ABSTRACT

Spartan Web Application Firewall

by Brian C. Lee

Computer security is an ongoing issue and attacks are growing more sophisticated. One category of attack utilizes cross-site scripting (XSS) to extract confidential data such as a user's login credential's without the knowledge of either the user nor the web server by utilizing vulnerabilities on web pages and internet browsers. Many people develop their own web applications without learning about or having good coding practices or security in mind. Web application firewalls are able to help but can be enhanced to be more effective than they currently are at detecting reflected XSS attacks by analyzing the request and response data sent between the web application by a user's browser to more quickly determine if a reflected XSS attack is being attempted. Spartan Web Application Firewall is designed to do this efficiently without being limited to requiring users to be using a specific web browser or web browser plug-in.

ACKNOWLEDGMENTS

I would like to thank Dr. Thomas Austin for providing guidance and expertise throughout this project, Dr. Stamp for getting me interested in computer security through the classes I have taken with him, and Dr. Chris Tseng for teaching me the basics of JavaScript and PHP.

TABLE OF CONTENTS

CHAPTER

1	Introduction	1
2	Background	3
2.1	Attacks	3
2.1.1	SQL Injection Attacks	3
2.1.2	XSS attacks	4
2.1.3	Denial of Service Attacks	7
2.1.4	Clickjacking	8
2.2	Defenses	8
2.2.1	Browser Add-on defenses	8
2.2.2	Browser Built-in Defenses	9
2.2.3	Content Delivery Networks	11
2.2.4	Web Application Firewalls	12
3	Spartan WAF	18
4	Testing Environment	28
5	Results	33
6	Conclusion and Future Work	38
	LIST OF REFERENCES	40

LIST OF FIGURES

1	Login form on Chrome	29
2	XSS detection on Chrome	29
3	Login form on Firefox	29
4	Reflected XSS on Firefox	29
5	Forum with initial posts	30
6	Forum posts in MySQL	31
7	Second to last post in forum thread	31
8	Last post in forum thread	31
9	Firefox Login Test with Spartan Web Application Firewall Active . .	33
10	Page Source for Firefox Logic Test with Sanitized Output	33
11	Elapsed time in seconds for non-matched rules by id sorted by elapsed time in seconds, scaled-up testing	34
12	Elapsed time in seconds for matched rules by id sorted by elapsed time in seconds, scaled-up testing	36

CHAPTER 1

Introduction

Because of the wide availability of web development tools, the ease of setting up a web site, the increasing usage of the internet, the convenience of accessing data, and the heterogeneous nature of websites, web security is an increasingly necessary field. Web security is constantly evolving and many methods were made to prevent unauthorized access to sensitive information such as personal, financial, and account data.

An increasing number of attacks on big profile targets such as corporations demonstrate the inadequacy of defenses that were put in place for protecting valuable data. Each major data breach causes not only the leak of sensitive customer data, a decrease in consumer confidence, and a hit to the victim corporation's reputation, but also intellectual property that corporations rely on to sustain or grow their revenue. In some cases, it only takes one vulnerability to be discovered and exploited by attackers to compromise an entire network.

These unauthorized accesses can and often do occur from poor programming techniques that are exploited by malformed input data. There are many layers that require security, starting from the browser and going all the way to the server, including any intermediate layers such as firewalls. Many web applications are not designed with security in mind and many programmers lack knowledge of common web security practices. Web application firewalls are built to address this issue and offer security to a wider audience than security built into web browsers. They do this by adding another layer of defense between a potential attacker and a web application to

prevent malformed input data from affecting a web application or being passed back from the web application to the browser by sanitizing or rejecting requests containing suspicious input data.

CHAPTER 2

Background

2.1 Attacks

There are multiple categories of attacks, each one with potentially devastating effects. Defending against these different attacks require security to be tailored to a specific type of attack. However, some attacks such as stored cross-site scripting attacks and SQL injection attacks, described below may be similar enough to defend against both at the same time.

2.1.1 SQL Injection Attacks

Structured Query Language, also known as SQL, is a language that is used to interface with databases. Many web applications rely on databases to store a multitude of data, ranging from user credentials, content, and statistics. For example, banks typically store all of the account information about members in various databases. Banking websites will need to use SQL query for both user credentials as well as the account information. Because SQL is a common language interface for many databases, it is a big target to use as an attack vector for malicious attackers.

SQL injection attacks target databases in order to steal or corrupt data. The way that this attack is typically carried out is by exploiting known vulnerabilities, such as unsanitized user input, that is put into part of a query and directly affects the operation. For example, a basic login page will take credentials like user ID and password and build a SQL query to verify them. If either field is unsanitized, meaning that no checking for potentially harmful input is done, the vulnerable web application can allow unauthorized operations to be run. An attacker can then get

access to read entries, or in the worst case write, delete, alter, or change the structure of the database.

Consider the following PHP code snippet which builds a SQL query to pass to the database: `SELECT * FROM USERS WHERE userID=$userid AND Password=$password;`

In the above SQL statement, the `userid` can be replaced with `' OR '1'='1';`, which would result in every field in the table `USERS` to evaluate to `true`. What this attack essentially does is trick the web application into authorizing a potential attacker regardless of whether the `userid` or `password` is correct. The statement could even be followed up with potentially destructive SQL statements with the rest of the original statement commented out. If the database were designed with a potential attack such as this in mind, the web application designer could take precautionary actions such as sanitizing the input to prevent SQL injection attacks from having ill effects.

2.1.2 XSS attacks

There are two categories of cross-site scripting (XSS) attacks: reflected and stored. Reflected XSS attacks use known vulnerabilities of legitimate web application in order to execute unauthorized scripts and are generally targeted at individual users, while stored XSS attacks are able to make changes to websites themselves and can affect any user visiting the compromised web site.

2.1.2.1 Reflected XSS Attacks

Reflected XSS attacks are not only the most common form of cross-site scripting attack, but are also the most common security exploit according to OWASP [1].

Vulnerabilities in web applications are typically caused by design flaws in which some input was not validated nor sanitized. These vulnerabilities are typically found by attackers trying various forms of input to attempt to run a short script. Vulnerable web servers may even be listed in a repository used by malicious users and available to attackers to exploit.

This form of attack typically relies on social engineering to trick victims; users may receive emails seemingly from friends, family, or a company they do business with containing injection scripts on vulnerable pages or click on links found from a web site. Some victims may even receive emails that appear to have been sent from a government agency, usually requiring some sensitive personal information, which creates a sense of great urgency in order to get victims panicked, preventing them from thinking more clearly. These attacks are usually carried out with the intent to steal information such as account credentials or perform unauthorized requests on behalf of a compromised user account.

As an example of a reflected cross-site scripting attack, consider the following (fabricated) phishing URL that is sent to a victim through an email prompting them to log into their account for an urgent reason:

```
http://www.vulnerablesite.com/welcome&username=<script>alert(‘‘There  
is a problem with your account. Please visit  
http://www.vulnerablesite.badsite.com and log in again.’’);</script>
```

and assume that the vulnerable webpage contains the PHP code snippet:

```
echo ‘‘Welcome $username!’’
```

The result is that although the victim is first taken to the vulnerable webpage, they are immediately prompted to visit the attacker’s website and attempt to log in there. The attacker can then obtain access to the victim’s login credentials.

Even news websites are vulnerable to attacks. Tom's Guide identifies articles on the New York Times website dated before 2013 have a known vulnerability to reflected XSS attacks [2]. No reports seem to have been made regarding any attacks exploiting this vulnerability but they are still potential targets for future attacks.

2.1.2.2 Stored XSS Attacks

Stored XSS attacks require a compromised web server in which malicious scripts are stored in the web server itself and included in web sites that it is serving. This is a more severe form of attack because it affects all users to a website, not just those targeted using social engineering. It can be harder to detect because third party websites are not necessarily involved; the script can reside entirely on the web server.

One common vector for stored XSS attacks is sites which have user-generated content, such as social networking sites and forums. Vulnerable web sites might not sanitize the user-submitted content, leading to third party scripts to execute on users just viewing the site. This can be difficult to detect since this type of website can be expected to have third-party content, and differentiating good or bad content may be challenging.

Another potential source for a stored XSS attack is through advertisements. Internet advertisements are often included using `<script>` tags since isolating them in separate iframes makes advertisement targeting impossible [3]. Malicious advertisements can access data in the web application and exfiltrate it to an attacker's server, or utilize it to try different vulnerabilities until one is found, allowing additional malware to be installed. This form of attack has even affected Google's DoubleClick advertising services [4], which is used on roughly 80% of websites that offer advertising [5].

2.1.2.3 mXSS Attacks

Hederich introduced a new type of XSS attack entitled *mutation-based XSS*, or *mXSS* [6]. This type of attack utilizes `innerHTML`, which is an HTML DOM property which can be used to set or retrieve the HTML content of an element, and other properties. Hederich reports that the top three browsers, Firefox, Chrome, and Internet Explorer, are all vulnerable to mXSS attacks. mXSS attacks take advantage of the browser layout engine to bypass XSS filters and becomes an active XSS attack vector. One example of a type of site that would use `innerHTML` and thus be vulnerable are mail clients such as Yahoo! Mail.

2.1.3 Denial of Service Attacks

The intent of denial of service (DoS) attacks are to essentially bring a web server offline by exhausting the resources on that web server. This is accomplished by opening many connections to the web server from one or a few sources until most or all connections are used. Future connections are held up waiting and time out.

Another flavor of this attack is a distributed denial of service (DDoS) attack in which the connections being opened come from many different sources, making it harder to defend against. In addition to this, the source IP addresses can be "spoofed", meaning fake, which prevents defending against this type of attack by blocking specific IP addresses. This sort of attack is typically carried out through botnets by infecting a large number of machines having them connect to a server where commands can be given [7].

2.1.4 Clickjacking

Clickjacking is a way to fool a user into clicking on something other than what the user intended to click on by using hidden elements by utilizing IFRAMEs to load a victim website and Cascading Style Sheets (CSS) [8]. One goal of clickjacking can be to generate advertisement revenue by tricking victims into viewing them advertisements hosted on malicious web sites [9].

2.2 Defenses

The main purpose of this project is to defend against reflected XSS attacks. Different types of defenses against reflected XSS attacks are already available. The most prevalent ones are built into the browsers themselves, and others are available through browser add-ons and web application firewalls. Common strategies for these defenses are checking for specific regular expressions, checking the request and response data, sanitizing data, and checking for third party scripts.

2.2.1 Browser Add-on defenses

Since there are multiple browsers in use today, different browsers have different add-ons but many of the popular ones (including ones for security) are available in some form on each of the popular browsers.

2.2.1.1 NoScript

NoScript for the FireFox browser is one of the most popular security-oriented browser add-ons. By default, it blocks all Javascript, Flash, and Java with the exception of websites that are whitelisted [10]. However, NoScript is very customizable and allows for different actions to be taken, such as temporarily (or permanently)

allowing scripts to run on blocked websites. This gives the user a lot of control over what can be executed on a given website. NoScript is designed to detect Javascript injection of whitelisted sites and flag them as a possible reflected XSS attack. By default, even requests from one whitelisted website to another will be checked by NoScript's InjectionChecker engine which "sanitizes only requests which contain conspicuous fragments of HTML or syntactically valid JavaScript" [10].

NoScript's approach is very effective, but introduces many false positives by sanitizing request data instead of response data and does not confirm that the sanitized data appears in the response [11]. Many internet users are not familiar with the concept of whitelisting specific scripts and as a result, may just disable or globally allow scripts to get their websites to work which defeats the purpose of this add-on.

2.2.2 Browser Built-in Defenses

Several web browsers have started to include built-in defenses to protect users against reflected cross-site scripting attacks. Having built-in defenses at the browser level allows for actions to be taken after parsing the HTML, which can be beneficial in eliminating false positives and catching attacks that can rely on encoding. In addition to this, due to the fact that they can access data after it is parsed, the data being checked is what will be displayed, increasing the accuracy while also decreasing the overhead of having to do parsing twice like some other defenses.

2.2.2.1 Internet Explorer 8

Microsoft had introduced protection against reflected XSS attacks in Internet Explorer by parsing the request and response information and searching for patterns, such as `<script>` tags, and sanitizing the page accordingly.

However, the way that the detection itself worked caused problems and even created additional vulnerabilities. One of the problems was with false positives: if a user were executing a search passing in `<script>` tags, the browser will assume that a reflected XSS attack is being executed and sanitize the page accordingly. Furthermore, this security feature introduced new vulnerabilities - namely by inducing a false positive to mangle security-oriented code. Bates provided the following example of a url that would prevent a security script from being loaded [12]:

```
http://victim.com/?<script src="secure.js"></script>
```

Because the script is contained in both the request and response, the browser will incorrectly identify it as a reflected cross-site scripting vector and will mangle the script.

One real-world example shown on stackoverflow is a search query on Microsoft's Bing search engine that generates script errors and can break security-oriented code [13]:

```
http://www.bing.com/search?q=%3Cscript+type%3D%22text%2Fjavascript%22%3E
```

Another of these new vulnerabilities is that Internet Explorer 8 “does not correctly approximate the byte-to-character decoding process” [12]. This allows for web pages that do not specify a character set to be vulnerable against an attack that uses UTF-7 encoding. According to a blog post from ZDNet, Microsoft had posted a response to reports of vulnerabilities in IE8's reflected XSS security and released a security patch MS10-002 [14]. Unfortunately, it seems that the security patch did not address all of the vulnerabilities because of the perceived tradeoff between security and compatibility [15] and can still expose certain web sites to these vulnerabilities.

2.2.2.2 Chrome

Google’s Chrome browser utilizes an implementation of the XSSAuditor architecture, which resides between the browser’s HTML parser and Javascript engine, allowing better performance with a higher degree of accuracy compared to the traditional method of checking the request and response data prior to parsing [12]. In order to have a high degree of accuracy in terms of reducing false negatives as well as false positives, checks for attacks should consider how data, such as text written in unicode, is represented in the browser. However, to do so prior to the actual parsing would incur a higher performance cost due to the fact that it has to effectively parse the data twice. By placing the security module inbetween the parser and the Javascript engine, it is able to selectively view only scripts that are going to be run, saving the performance cost of both an initial parsing as well as suspicious but benign patterns that are not actually scripts.

2.2.3 Content Delivery Networks

Content Delivery Networks (CDNs) such as as CloudFlare [16] and Akamai [17] provide several services, including security-oriented security features, to web application developers. One of the main purposes of CDNs is providing multiple servers to cache data to improve performance for users, reduces bandwidth to the web application server, and greatly reduces the risk of the web application server from going down (by somewhat eliminating the single point of failure aspect). Caching data and having servers localized in different areas allows for users to have reduced response times in terms of loading pages for web applications due to the relatively close proximity of the CDN servers. Since data is cached on the CDN servers, it allows for reduced bandwidth to the web application server itself. Finally, it helps defend

against DoS/DDoS attacks by masking the true address of the web application server and using the different servers for redundancy.

In addition to this, CDNs can take additional steps to detect and defend against DoS/DDoS attacks. There are multiple operating modes in which requests are filtered, ranging from normal (allowing everything) to paranoid (only allowing trusted connections) mode.

2.2.4 Web Application Firewalls

A web application firewall (WAF) is a layer of security which applies a set of pre-defined rules against requests coming in and responses going back from a web application. WAFs are typically used as a defense against common attacks such as XSS attacks and SQL Injection attacks [1].

However, web application firewalls do have weaknesses. Schmitt and Schinzel describe two different types of side channels, which are described as “unintentional and hidden communication channels that appear if the publicly observable behavior of a process correlates with sensitive information” [18]. These two types of side channels are categorized as timing side channels, and storage side channels. Their focus was on timing side channels in which they measure the response time for different requests to try to ascertain information about not only what gets flagged by the web application firewall, but also which web application firewall is in use based on fingerprinting. They describe three different web application firewall network topologies: a stand-alone web application firewall, a web application firewall that is included as a plug-in such as ModSecurity, and a web application firewall that is included as a programming library in the web application itself. Using ModSecurity as an example, they can determine and choose which rules are in effect and what actions are taken for each one, i.e.

whether a request is passed on to the web application or blocked. Their assumption was that the response time for a request that was blocked by a web application firewall would take a noticeably shorter amount of time than one that was passed on to the web application. Their experiment was successful and they found that response times are not typically normalized when it comes to web application firewalls, and that this sort of attack cannot be easily defended against if the attack is distributed across many users.

2.2.4.1 ModSecurity and OWASP

ModSecurity is an open-source Web Application Firewall that works on several platforms, such as Apache Server. It offers a variety of features, such as different levels of HTTP logging, three different security models, and a flexible rule engine [19]. HTTP logging at different levels allow for HTTP requests and response to be logged as the user desires, up to full logging of every transaction along with the timestamp. The three security models are:

- Negative Security Model
- Rule-based Security Model
- Positive Security Model

In the negative security model, ModSecurity will give an anomaly score for each request, IP address, application sessions, and user accounts, which can be logged or rejected if the anomaly score is high enough. In the rule-based security model, ModSecurity utilizes different sets of rules, namely the OWASP ModSecurity Core Rule Set which defends against common web application attacks and is maintained by an active userbase. Users can write and upload their own rules to be included

along with a description of what the rules defend against, allowing the core rule set to protect against recent attacks. The positive security model can be seen as a very restrictive approach, only allowing whitelisted requests and rejecting all others.

Because ModSecurity is open-source, web application developers have a great deal of customization such as the logging of metadata and level of security desired. Likewise, the rules listed on the Open Web Application Security Project (OWASP) are very customizable, allowing users to select only a subset of rules to use, modify the security level of a rules, modify the action taken by a rule, or modify the rule itself. Below is an example of a rule included in the OWASP core rule set `modsecurity_crs_41_xss_attacks.conf` [1] which searches for script tags and logs any matches as a critical anomaly:

```
SecRule ARGS "(?i)(<script[^\>]*>[\s\S]*?</script[^\>]*>|<script[^\>]*>[\s\S]
    ]*?</script[[\s\S]]*[\s\S]|<script[^\>]*>[\s\S]*?</script[\s]*[\s]|<
    script[^\>]*>[\s\S]*?</script|<script[^\>]*>[\s\S]*?)" "id:'973336',phase
    :2,rev:'1',ver:'OWASP_CRS/2.2.8',maturity:'1',accuracy:'8',t:none,t:
    urlDecodeUni,t:htmlEntityDecode,t:jsDecode,t:cssDecode,log,capture,msg:'
    XSS Filter - Category 1: Script Tag Vector',tag:'OWASP_CRS/WEB_ATTACK/XSS
    ',tag:'WASCTC/WASC-8',tag:'WASCTC/WASC-22',tag:'OWASP_TOP_10/A2',tag:'
    OWASP_AppSensor/IE1',tag:'PCI/6.5.1',logdata:'Matched Data: %{TX.0} found
    within %{MATCHED_VAR_NAME}: %{MATCHED_VAR}',severity:'2',setvar:'tx.msg
    =%{rule.msg}',setvar:tx.xss_score=%{tx.critical_anomaly_score},setvar:tx
    .anomaly_score=%{tx.critical_anomaly_score},setvar:tx.%{rule.id}-
    OWASP_CRS/WEB_ATTACK/XSS-%{matched_var_name}=%{tx.0}"
```

The above rule has a rule id of 973336, and `phase:2` specifies that ModSecurity will run the rule against the request body. In fact, all of the XSS injection rules

run on the request body. The five phases for Apache requests from the ModSecurity wiki [20] are:

- Request headers
- Request body
- Response headers
- Response body
- Logging

The fact that these rules run only on the request body can create false positives, but are effective in defending against stored cross-site scripting attacks on web applications that do not otherwise have sufficient checking of input. Accuracy varies from 1 to 9 where 1 generates many false positives and 9 is very accurate. Maturity indicates the level of testing that the rule has undergone along with the length of time it has been available. Various other options assigns tags, specifies options for logging, actions taken, and sets the severity of any matches. In general, this rule searches for various combinations of `<script>` tags along with other characters indicating a reflected cross-site scripting attack.

A study was done by Canali, Balzarotti, and Francillon [21] regarding the effectiveness of not only the ModSecurity module, but also the OWASP core rule set. They had different attacks and saw varying levels of success from ModSecurity:

- SQL injection attacks
- remote file upload with code injection using web shell

- remote file upload of a phishing kit
- IRC bot activity
- uploading of known malware.

ModSecurity with the OWASP core rule set was found to be completely to somewhat successful at blocking SQL injection attacks and code injection attacks using web shell, and successful at logging SQL injection attacks, code injection attacks using web shell, and remote upload of phishing kits. However, it had no success at all in detecting suspicious IRC bot activity and uploading of known malware.

2.2.4.2 TokDoc

Kreuger proposed another type of web application firewall that utilized anomaly detection to decrease the number of false negatives from other web application firewalls such as ModSecurity [22]. TokDoc goes through all of the relevant data for the path, GET and POST, and header fields. Based on the usage, Krueger identified four different types based on the properties of the fields:

- constants which never change
- enumerations which has a small set of values
- machine input which contains metadata
- human input that is widely variable.

By categorizing the data fields by these four types, some general assumptions can be made for any given data field. After running anomaly detection on the data fields, subsequent healing actions can be taken [22]:

- dropping the token, meaning do not pass it along to the web application
- preventative encoding which translates special characters for HTML and SQL
- replacement with the most frequent normal value
- replacement with the nearest value

The results were promising for detecting false negatives, however it is not clear how effective the replacement healing actions are. Although for many cases such as typos it may be desirable, there may be some legitimate anomalies where a healing action would be undesirable.

CHAPTER 3

Spartan WAF

An implementation of a web application firewall named “Spartan WAF” was created to bring browser-style reflected XSS defenses to web applications. This web application firewall is different than existing solutions (such as ModSecurity) in that it compares both the request and response data in order to reduce the number of false positives. Here’s an overview of how it works: First, the web application includes the web application firewall in the form of a PHP script. The input to the web application (such as form data) is then passed through the web application firewall, which runs one or more rule files against the input. The input rule files are compatible with the existing Open Web Application Security Project (OWASP) rules; however, they mainly utilize the regular expressions provided but do not take the same actions. Instead, for any input matching a regular expression in the rule files, it will create a temporary file and save it, recording not only the input, but the rule id that it was matched to. Once the web application firewall processes the input, the web application then sends the input along to the web server. After the web server passes the response back to the web application, the web application firewall will do another scan with any response data that is displayed on the page. It will again run the response data against the sets of rules in the same rule files and compare any matches with those recorded in the temporary file. If there are matches in both the request and response data, the data is then flagged, recorded, then sanitized to prevent any ill effects.

An example of a false positive that would trigger ModSecurity running with the OWASP XSS rules is a website, such as a tech blog, that has a search function. A

user wishes to search for examples of `<script>` tags and types it into the search bar. Even if the web application sanitized the input prior to referencing it, ModSecurity would detect the `<script>` tag and take action. However, with Spartan WAF, the `<script>` tag would be passed along to the web application. The web application would sanitize it, run the search, then print out any results along with the sanitized search string. Spartan WAF then sees the sanitized search string and will not take action. Even if the web application did not sanitize the search string and returned the original search string back, Spartan WAF would then sanitize it accordingly.

One of the drawbacks of this approach is that `echo`, which basically generates HTML that can be interpreted by the web browser, is a PHP language construct and not a function so it cannot be overridden. However, Spartan WAF includes a preprocessor portion that converts a web application to use a separate function to replace calls to `echo` with a similar function `respEcho` which compares HTML generated by the web application to input data that was previously flagged as suspicious. In other words, this function ensures that any information passed along from the web application to the browser will go through the web application firewall.

The advantages of this implementation lie in the layer in which the web application firewall resides. Since it is part of the web application firewall itself, it does not depend on the web browser that is accessing the data, nor the platform. Also, because it compares both the request and response data, it can reduce the number of false positives that are typically flagged. Due to the simplicity of the usage, it can be applied to web applications with some minimal effort to detect and prevent attacks with readily available proven rule files.

In regards to the timing side channel in which one can infer actions taken by the web application firewall based on the response time, as far as reflected XSS attacks

are concerned, this attack is not effective. The reason for this lies in the design of Spartan WAF. Because any request data that matches any rules configured is saved then compared to the response data, the action taken is done after the web application has returned the data. The differences in measureable response times should be only noticeable depending on the rule matching algorithm and the comparison between the request and response data. Thus, an attacker will not be able to easily determine what action was taken.

Because the web application firewall uses the same rules as OWASP, the detection rate should be identical. The main goal of the test is to prove its usefulness in detecting false positives and measure the overall performance of the web application firewall, in particular which rules have a higher performance cost and their overall effectiveness.

To use Spartan Web Application Firewall, first the script must be included and the rule files with extension `.conf` must be in the same directory as `waf.php`:

```
include "waf.php";
```

then a call must be made to check the input against the rule files:

```
checkInput($_GET);
```

The function `checkInput` expects an array of length 1 or greater. It goes through each of the input array elements and calls another function `runRules` with the key of the input element along with current input element, which contains the main functionality of Spartan Web Application Firewall. For example, if the `$_GET` array is passed in, the keys for fields it contains can include `$username` and `$password`.

```
function checkInput($input) {  
    // make sure there is input data
```



```

if (!empty($input)) {
    // loop through all input data
    while ($value = current($input)) {
        // run rules on each
        runRules(key($input), $value);
        next($input);
    } // end while
} // end if
} // end function checkInput

```

The function `runRules` takes in one key value and one input value.

```
function runRules($key, $input)
```

The main reason for the key value to be passed is to allow log files to show which input field contained the detected reflected cross-site scripting attack vector. The start of the code will first open both `perfdata.txt` for output for recording performance data and `input_matches.txt`. For each, it will create the files if they do not exist, or append to the end if they do exist.

```

// create or open perfdata.txt which contains performance data
$perfhandle = @fopen("./" . "perfdata.txt", "a") or die ("couldn't open the
    file");

// create or open input_matches.txt
$inputhandle = @fopen("./" . "input_matches.txt", "a") or die ("couldn't
    open the file");

```

Once the `input_matches.txt` temporary file is opened, it first checks to see if the input value had already been checked and matched. To do this, it utilizes the

Unix/Linux `grep` function to see if the input data exists in the `input_matches.txt` file.

```
// if the input was already matched, do not check again
if( exec('grep '.escapeshellarg($input).' ./input_matches.txt')) {
    return;
} // end if
```

Afterwards, it starts searching the current working directory for any `.conf` files, which are the rule files. For compatibility with the OWASP ModSecurity rules, all rule files must have the `.conf` file extension. It will open any `.conf` files for input only and start parsing each rule in each file.

```
// loop through all files in the current working directory
while (($entry = readdir($fd)) !== false) {
    // only read .conf files
    if ((substr($entry, -8) == ".conf")) {
        // open the conf file
        $handle = @fopen("./" . $entry, "r") or die ("couldn't open the file");
```

If the open is successful, it read one line at a time and remove any whitespaces from the beginning and end of each line. It then searches for the keyword `SECRULE` which each rule is prefaced with. Once the keyword is found, it will split the line using “`“` as the delimiter. The first element will be the keywords `SECRULE ARGS` and the second element will contain the regular expression.

```
// read lines one by one from the conf file
while (($buffer = fgets($handle, 4096)) !== false) {

    // trim extra white space from beginning/end of line
```



```

$buffer = trim($buffer);

// search for rules only, keyword SECRULE
if (strpos($buffer, "SECRULE") !== FALSE) {

    // split the line by double quotes
    $pattern = "/\\"/";
    $matches = preg_split($pattern, $buffer);

    // if not empty
    if ($matches) {

        // initialize, no hits
        $hit = "NO";

        // element 1 will contain the rule
        if (isset($matches[1])) {

            // copy element 1
            $j = 1;
            $result = $matches[j];

```

The rule identifier can either be on the same line or on the next line. PHP uses the Perl Compatible Regular Expressions (PCRE) library, which requires regular expressions to be enclosed with forward slashes /. The regular expressions in the ModSecurity rules are not enclosed in forward slashes, so they need to be added:

```

// make sure the rule is enclosed with forward slashes

```



```

if (substr($result,0,1) !== "/") {
    $result = "/" . $result;
} // end if

if (substr($result,-1,1) !== "/") {
    $result = $result . "/";
} // end if

```

PCRE also requires certain characters such as + must be escaped with a backslash \, so Spartan Web Application Firewall will escape those characters:

```

$result = str_replace("+", "\+", $result);
$result = str_replace("-", "\-", $result);
$result = str_replace("'", "\'", $result);

```

After the regular expression is parsed and PCRE compatible, the timestamp is recorded in microseconds in order to measure the performance of a given rule before matching the regular expression to the input. If a match was found, the matching input data value is written out to a temporary file. However, if the input was already The ending elapsed time is recorded in microseconds and the results are written out if there were no errors in the regular expression.

```

$elapsed = 0;
if (preg_match_all($result, $input, $out) !== FALSE) {
    $endTime = microtime(true);
    $elapsed = $endTime - $startTime;
    if (count($out) !== 0) {
        $hit = "YES";
    }
}

```



```

        // write the matched input into the file
    else {
        fwrite($inputhandle, $input . PHP_EOL);
    } // end else
} // end else if
} // end if

// write performanceresults out
if ($elapsed > 0) {
    fwrite($perfhandle, "$ruleid,$elapsed,$hit" . PHP_EOL);
}

```

After all of the input values are checked against the rule files, all matched input values are recorded in `input_matches.txt`. The web application then calls `respEcho` to generate the HTML to be passed back to the user's web browser. For each HTML string, it will check to see if it contains any of the matched input. To accomplish this, it will open the `input_matches.txt` file, read each line, strip out the whitespaces at the start and end of each line, then search the input for each of the strings. If any matches are found, it will call `str_replace` to replace all instances of that match within the string with `htmlspecialchars` to sanitize that particular pattern only. This way it will not affect any other unmatched code which may therefore mangle too much of the HTML. Function `respEcho`

```

function respEcho($string) {

    // open file containing matches found for input
    $inputhandle = @fopen("./" . "input_matches.txt", "r") or die ("
        couldn't open the file");

```



```

// read a line
while(($buffer = fgets($inputhandle, 4096)) !== false) {

    // strip whitespace from beginning & end
    $buffer = trim($buffer);

    // search for a match
    if (strpos($string, $buffer) !== false) {
        //match found, sanitize any matches
        $string = str_replace($buffer,htmlspecialchars($buffer)
            ,$string);
    }
}

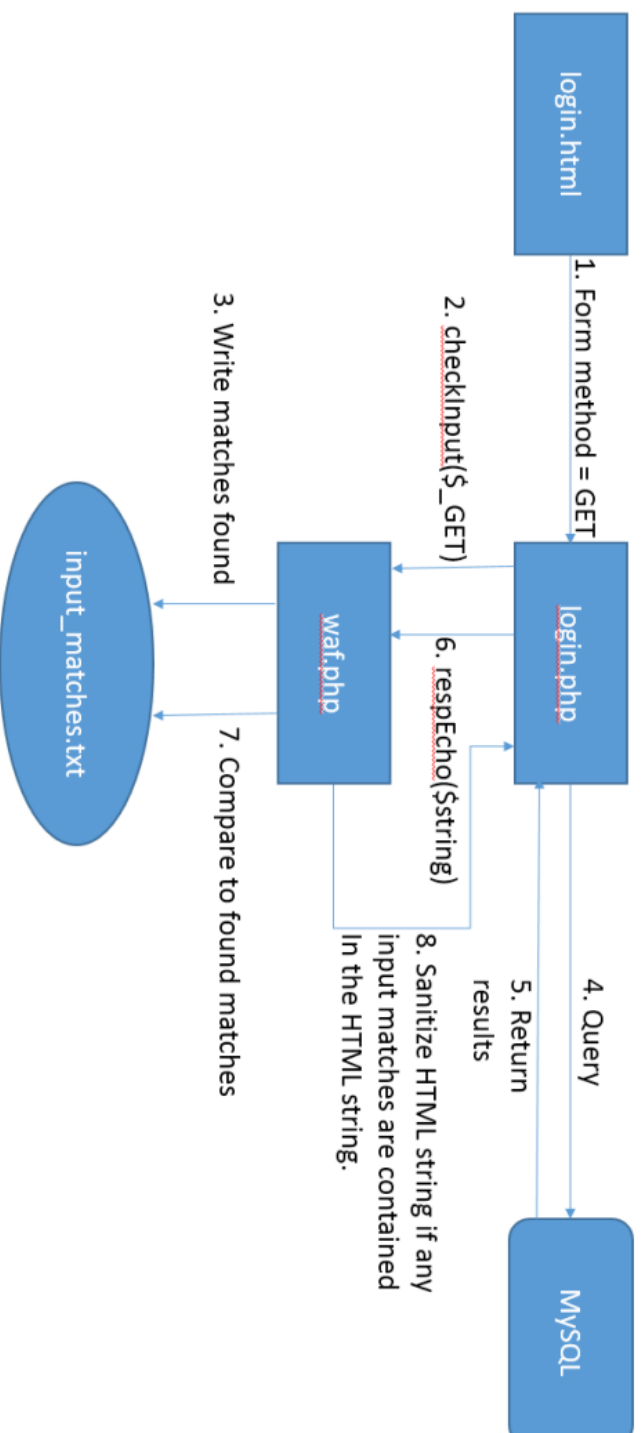
// do the echo
echo $string;

// close the file
fclose($inputhandle);
}

```

The overview of how Spartan Web Application Firewall is illustrated in Figure 3.

Figure 1. Spartan Web Application Firewall Overview



CHAPTER 4

Testing Environment

The testing was done on a Linux-based machine running on an Ubuntu derivative named Kubuntu [23] using Apache server version 2.4.7 [24] as the web server , MySQL version 14.14 [25] as the backing database, PHP 5.5.9 [26] for the web applications and web application firewall, phpMyAdmin 4.0.10deb1 [27] for a graphical user interface to the MySQL database, and no-ip [28] to provide dynamic dns service for remote access. The web browsers used for testing were Google Chrome 38.0.2125.111 (64-bit) [29] and Mozilla Firefox 33.0 [30]. The initial test vector consisted of simple scripts that displayed an alert.

The prototype testing platform was a simple login page which was coded in HTML and PHP with user credentials stored in a MySQL database. The login webpage included the web application firewall script and ran the rule checking against the request and response data with no sanitization to see the results of any reflected scripts. As expected, different results were seen between Chrome and Firefox. Chrome appeared to have detected the reflected script, which can be seen when looking at the page source in Figure 1 on page 29 and Figure 2 on page 29 below. The `alert("1");` is highlighted in red and the tooltip says "Token contains a reflected XSS vector". In comparison, Firefox had no such detection, as can be seen in Figures 3 and 4 on page 29 below, and the reflected XSS vector took effect. For ease of testing, Firefox is used exclusively for further testing the effectiveness of the web application firewall.

The next testing platform is a forum which is a fairly common sight on the internet. An open-source implementation called phpBB version 3 [31] was used to

Figure 1. Login form on Chrome

User id:

Password:

Not a current user? Register [here](#)

Figure 2. XSS detection on Chrome

```
<html>
<body>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />login
failed. userid <script>alert("1");</script> not found or password 1234567890
invalid. click <a href="index.html">here</a> to re-enter credentials
</body>
</html>
```

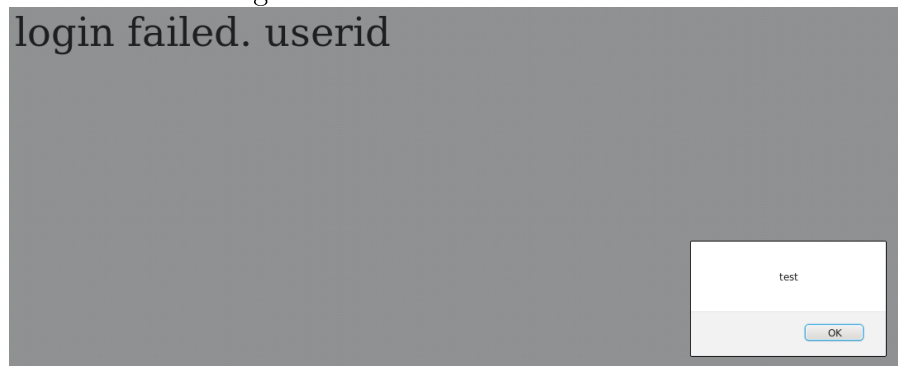
Figure 3. Login form on Firefox

User id:

Password:

Not a current user? Register [here](#)

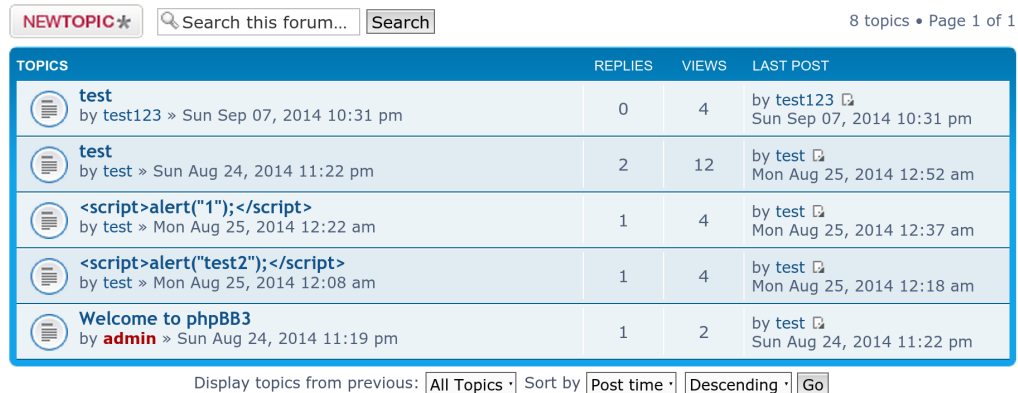
Figure 4. Reflected XSS on Firefox













achieve this, which uses MySQL to store content as well as metadata. The default configuration of phpBB did not allow for cross-site scripts to be propagated past the

initial submission. It utilizes functions such as the PHP function `htmlspecialchars`, which replaces HTML special characters such as `<`, `"`, and `>` with HTML entities `<`, `"`, and `>` respectively, to first sanitize the input before using it. This proved an effective defense particularly against stored cross-site scripting attacks. The code was modified to remove sanitization of input when creating new threads or posting replies to illustrate how both reflected and stored cross-site scripting attacks can be carried out and defended against. After the modification, the forum was populated with several new user accounts along with some initial posts as seen in Figure 5 below.

Figure 5. Forum with initial posts



TOPICS	REPLIES	VIEWS	LAST POST
 test by test123 » Sun Sep 07, 2014 10:31 pm	0	4	by test123  Sun Sep 07, 2014 10:31 pm
 test by test » Sun Aug 24, 2014 11:22 pm	2	12	by test  Mon Aug 25, 2014 12:52 am
 <script>alert("1");</script> by test » Mon Aug 25, 2014 12:22 am	1	4	by test  Mon Aug 25, 2014 12:37 am
 <script>alert("test2");</script> by test » Mon Aug 25, 2014 12:08 am	1	4	by test  Mon Aug 25, 2014 12:18 am
 Welcome to phpBB3 by admin » Sun Aug 24, 2014 11:19 pm	1	2	by test  Sun Aug 24, 2014 11:22 pm

Display topics from previous: All Topics Sort by Post time Descending Go

In Figure 6 below, the database is shown to have been populated with several posts. Some earlier posts show the effect of the sanitization with special character replacement as previously described. This allows the text to appear unchanged when represented on the website but prevents the potential ill-effects associated from being executed. The second post content was modified directly prior to the sanitization code being taken out to discover the extent of the sanitization, and the last two posts have unsanitized content as submitted by the user.

In Figures 7 and 8 below, the effects of the last two posts in Figure 6 can be seen. First, the dialog box from the second to last post is displayed, followed by the dialog

Figure 6. Forum posts in MySQL

←T→		post_id	topic_id	forum_id	poster_id	icon_id	poster_ip	post_time	post_subject
<input type="checkbox"/>	Edit Copy Delete	13	10	2	48	0	127.0.0.1	1408951328	<script>alert("1");</script>
<input type="checkbox"/>	Edit Copy Delete	8	6	2	48	0	127.0.0.1	1408950520	<script>alert("test2");</script>
<input type="checkbox"/>	Edit Copy Delete	14	10	2	48	0	127.0.0.1	1408952249	Re: <script>alert("1");</script>
<input type="checkbox"/>	Edit Copy Delete	11	6	2	48	0	127.0.0.1	1408951093	Re: <script>alert("test2");</sc...
<input type="checkbox"/>	Edit Copy Delete	16	2	2	48	0	127.0.0.1	1408953147	Re: test
<input type="checkbox"/>	Edit Copy Delete	2	2	2	48	0	127.0.0.1	1408947744	test
<input type="checkbox"/>	Edit Copy Delete	17	11	2	49	0	127.0.0.1	1410154297	test
<input type="checkbox"/>	Edit Copy Delete	1	1	2	2	0	127.0.0.1	1408947551	Welcome to phpBB3

box from the last post. Viewing the source as can be seen in Figure 9, the content for both posts are the original scripts. These are prime examples of how stored cross-site scripting attacks work and how they can affect a large number of users.

Figure 7. Second to last post in forum thread

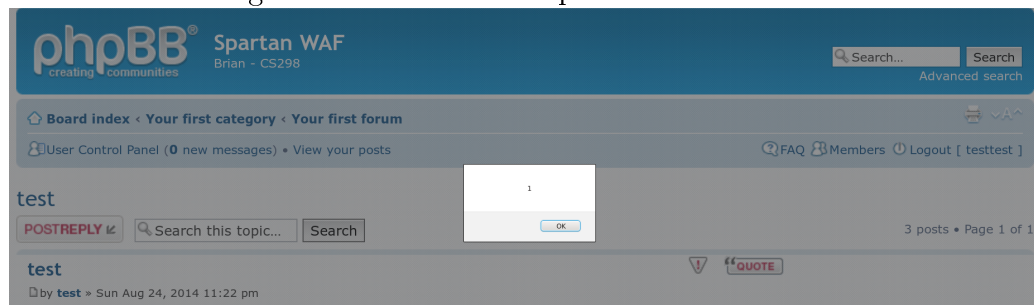
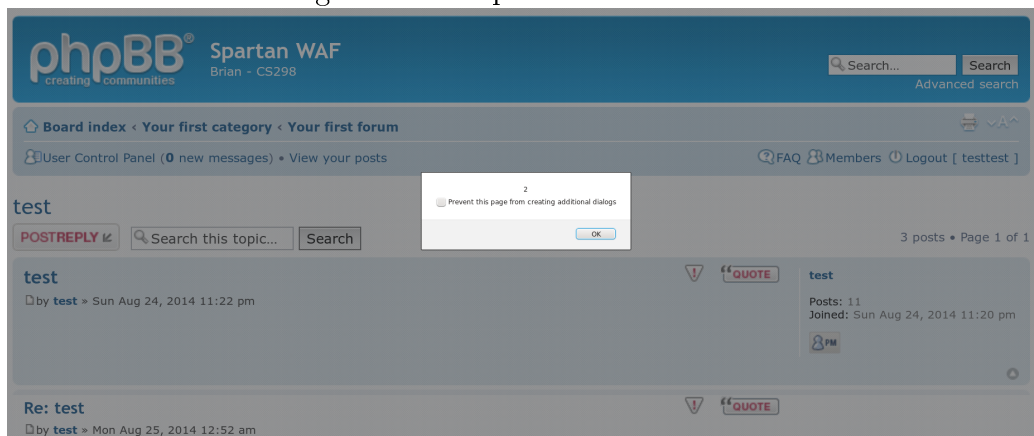


Figure 8. Last post in forum thread



Unfortunately, phpBB was not an ideal testing platform for Spartan Web Application Firewall. The reason for this is because it uses MySQL to save post content

even when previewing and it utilizes CSS to display content as opposed to the PHP language construct `echo`. However, it is a good test for ModSecurity since the rule checking is done on the request body only, and should prevent stored cross-site scripting attacks. As for detecting stored cross-site scripting attacks when the attack vectors are already stored on the web server, neither Chrome nor ModSecurity is successful. The reason for this is because it is difficult to tell valid scripts apart from malicious scripts, and any checking would introduce many false positives.

The final testing platform is to test a variety of different input data consisting of both “benign” and “harmful” inputs at a large scale in order to measure the performance of different rules. To accomplish this, a testing tool was created using Python version 3.4.2 [32] with the mechanize package [33] to send the data. The tool utilizes a file containing all of the input data and invokes the login page millions of times, each time sending a random combination of input. The rule id, elapsed time, and whether or not the rule matched is recorded for each rule by the web application firewall in an output file. The web application firewall runs every rule from all selected rule files once for each of the elements in the request data, and again for each of the elements in the response data. The performance data that was collected was then processed by additional python tools written to parse, sort, and average the results.

Because every rule is run each time the checking is done despite finding a rule match, the rules do not necessarily reflect the performance in a real-world environment. For this reason, a final testing platform stops comparing data to the rules once a match is found. This should allow us to ascertain the performance during an actual reflected cross-site scripting attack as well as which rules are matched the most often.

CHAPTER 5

Results

Now that the test scenarios for reflected cross-site scripting attacks were established, the next step was to test the effectiveness of the web application firewall. Although no real-world traffic data could be used for testing purposes, tests were done in which a mix of both “malicious” and “benign” input was passed in through the web application firewall. Referring back to Figure 4, it is apparent that the script was reflected and a dialog box with the word *test* is visible. Adding sanitization to the input before displaying the content in question allows the script to be shown without executing, as can be seen in Figure 9 below. Looking at the page source in Figure 10, it is apparent that the sanitization was effective. Although once a single match is found no more processing is required, the additional checking is done to get performance measurements of the different rules.

Figure 9. Firefox Login Test with Spartan Web Application Firewall Active

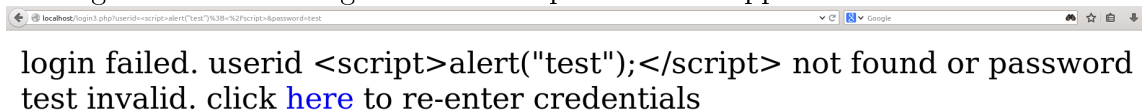


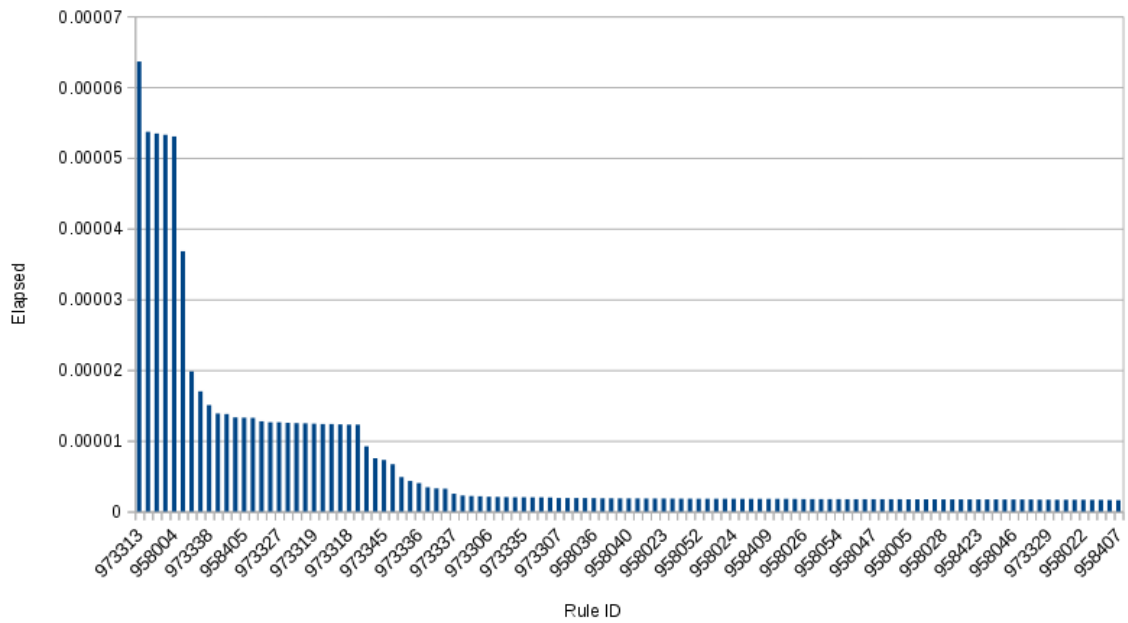
Figure 10. Page Source for Firefox Logic Test with Sanitized Output

```
1 <html>
2 <body>
3 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />login
  failed. userid &lt;script&gt;alert(&quot;test&quot;);&lt;/script&gt; not found
  or password test invalid. click <a href="index.html">here</a> to re-enter
  credentials
4 </body>
5 </html>
6
```

Preliminary test results did show a discrepancy in several of the rules in terms of the average performance, which was measured in microseconds.

After the testing was scaled up, several other rules showed an increase in processing time. Rearranging the results by elapsed time as can be seen in Figure 10, it is clear that the 6 rules that took the longest amount of time to process without getting a match are substantially larger than the others. 0.00002 seconds was chosen as a threshold for “acceptable” performance time. The 6 rules with the highest elapsed

Figure 11. Elapsed time in seconds for non-matched rules by id sorted by elapsed time in seconds, scaled-up testing



time in order of highest to lowest are:

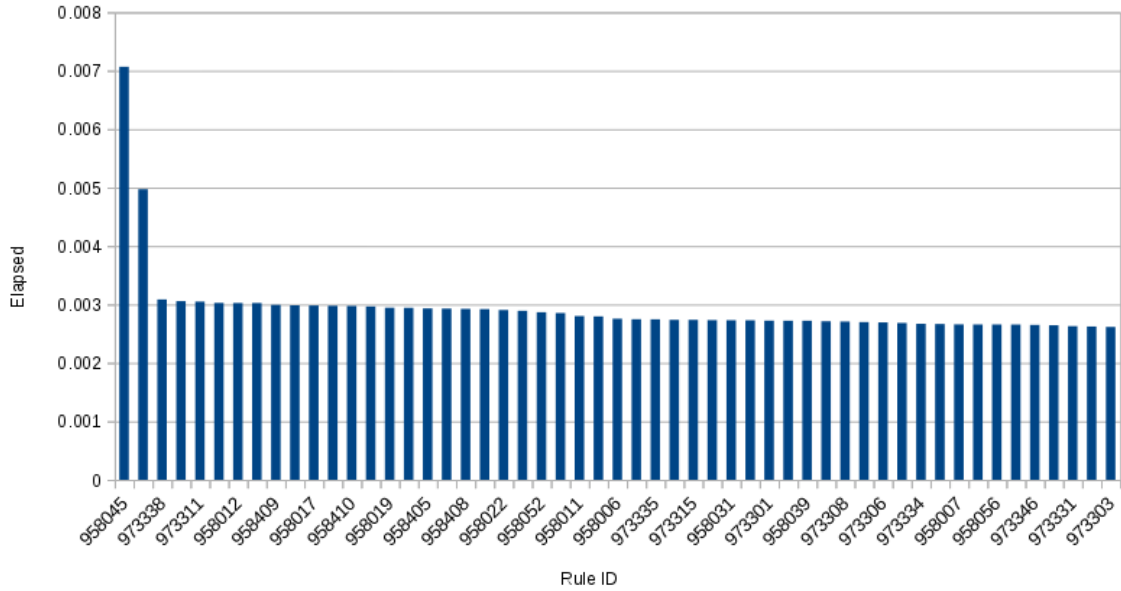
- rule 973313 which checks for the pattern “&{” to prevent “&{alert(‘xss’)}” alerts on Netscape 4, which seems to be an unreasonably high performance cost for an unsupported and dated browser. A cursory glance indicates that less than one percent of web traffic is using Netscape [34].
- rule 958045 which checks for url Javascript injection, which is a valid security risk.

- rule 958025 which checks for reflected cross-site scripting attacks using the HTML `lowsrc` tag with a script as a target.
- rule 958033 which checks for reflected cross-site scripting attacks using the HTML `src` tag with a script as a target. This differs from the previous rule 958025 since both are looking for a “complete word” that is either `lowsrc` or `src`, meaning that the `lowsrc` tag will not trigger the rule checking for the `src` tag.
- rule 958004 which checks for mXSS attack vectors using the HTML DOM property `“.innerHTML”` to allow for JavaScript code execution without explicit `<script>` tags [6].
- rule 981136 which checks for a multitude of JavaScript keywords such as `“onmove”` and `“onkeydown”`.

After analyzing all 6 rules, it is reasonable to leave out the rule with the largest performance cost but the others are arguably valuable in preventing reflected cross-site scripting attacks. Rule 958004 can be optional because the methods to inject JavaScript without utilizing `<script>` tags requires other keywords that would be detected by other rules, such as the `“onerror”` keyword being detected by rule 981136.

As can be seen in Figure 12, there are fewer results since not every rule was matched. The results for the rules that matched are generally more even at around .0025 to .0030 seconds each. The rule with the highest performance cost was 958045 which was previously identified as the second highest rule in performance cost when there was no match. The second highest was 973338 which searches for "XSS vectors making use of Javascripts URIs", which is also a good test to keep. The rest are fairly close to each other in performance. However, the performance of rules when

Figure 12. Elapsed time in seconds for matched rules by id sorted by elapsed time in seconds, scaled-up testing



detecting an actual attack does not matter much other than for preventing timing side channel attacks. The fact that the majority of the rules had similar performance, together with the fact that requests are always compared twice, should help minimize the effects of timing side channels.

There was one instance of performance data being evaluated on ModSecurity [35]. However, the results they observed did not contain enough information to compare the performance of different rules on ModSecurity and Spartan WAF. Not only were the elapsed times different, but also the ratio between rules such that one rule that was lower-performing than another was observed to be the opposite.

Keeping in mind that the data for the instances where there was no match is what is expected for benign data, the sum of all of the averages along with creating, writing to, reading from, and deleting a temporary file can be reasonably expected as overhead for the web application firewall for a given benign request. The average elapsed time

without Spartan WAF is 20042 microseconds, which is about 0.02 seconds. The average elapsed time with Spartan WAF active is 26445 microseconds, which is about 0.026 seconds. Although the increase in elapsed time is 32%, the overall increase would not be noticeable for normal users.

CHAPTER 6

Conclusion and Future Work

Spartan Web Application Firewall compares the request and response data and searches for matches, which reduces the amount of false positives that ModSecurity can have. The overall performance of the rules can be improved by removing high-cost low-impact rules such as rule id 973313 which protects against cross-site scripting attacks for users using the unsupported Netscape browser on vulnerable websites. However, the performance impact of Spartan Web Application Firewall is relatively low, adding less than a second of additional elapsed time, which is a fair tradeoff for the protection it offers.

Some areas that can be enhanced on the Spartan Web Application Firewall are:

- tie it closer to Apache, like ModSecurity, so it does not have to be called explicitly to check request and response data
- use of main memory instead of storing the input data in a file first in order to improve performance and avoid creating temporary files
- enhancing the handling of rules to allow additional customization of rules

In addition, although the web application firewall does have some natural defense against timing side channel attacks, by design it is not effective against SQL injection attacks nor stored XSS attacks. In order to further enhance the web application firewall, any SQL injection or stored XSS attack detection must be done prior to passing the data to the database. This can be accomplished by having bringing a context-awareness to the web application firewall. For example, SQL calls in PHP

can be overridden to first run the input against the rule set for SQL injection. Also, for common vectors for stored XSS attacks such as a forum, the payload, which can be a topic name or post body, can be enhanced by sanitizing the payload prior to making the SQL call.

LIST OF REFERENCES

- [1] “Cross-site Scripting (XSS) - OWASP”. Retrieved 11 Dec. 2014, from https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29
- [2] Scharr, J. “XSS Flaw May Exist on Old NY Times Article Pages”. Retrieved 11 Dec. 2014, from <http://www.tomsguide.com/us/xss-flaw-ny-times,news-19784.html>
- [3] Dong, X., Tran, M., Liang, Z., & Jiang, X. “Adsentry: comprehensive and flexible confinement of JavaScript-based advertisements”. *Proceedings of the 27th Annual Computer Security Applications Conference*, pp. 297–306, 2011.
- [4] Li, Z., Zhang, K., Xie, Y., Yu, F., & Wang, X. “Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising”. *Proceedings of the 2012 ACM conference on Computer and communications security*, pp. 674–686, 2012.
- [5] Gill, P., Erramilli, V., Chaintreau, A., Krishnamurthy, B., Papagiannaki, D., & Rodriguez, P. “Follow The Money Understanding Economics of Online Aggregation and Advertising”. *Proceedings of the 2013 conference on Internet measurement conference*, pp. 141–148, 2013.
- [6] Heiderich, M, Schwenk, J., Frosch, T., Magazinius, J., & Yang, E.Z. “mXSS attacks: attacking well-secured web-applications by using innerHTML mutations”. *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pp. 777–788, 2013.
- [7] Mao, Z. M., Sekar, V., Spatscheck, O., Merwe, J., & Vasudevan, R. “Analyzing Large DDoS Attacks Using Multiple Data Sources”. *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pp. 161–168, 2006.
- [8] Tang, S., Dautenhahn, N., & King, S. “Fortifying web-based applications automatically”. *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 615–626, 2011.
- [9] Shahriar, H., Devendran, V.K., & Haddad, H. “Proclick: a framework for testing clickjacking attacks in web applications”. *Proceedings of the 6th International Conference on Security of Information and Networks*, pp. 144–151, 2013.
- [10] “Noscript - JavaScript/Java/Flash Blocker for a safer Firefox experience! - what is it? - InformAction”. Retrieved 11 Dec. 2014, from <http://noscript.net/>

- [11] Pelizzi, R., & Sekar, R. “Protection, usability and improvements in reflected XSS filters”. *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, p. 5, 2012.
- [12] Bates, D., Barth, A., & Jackson, C. “Regular Expressions Considered Harmful in Client-Side XSS Filters”. *Proceedings of the 19th international conference on World wide web*, pp. 91–100, 2013.
- [13] “Internet Explorer 8 - IE8 XSS filter: what does it really do? - Stack Overflow”. Retrieved 11 Dec. 2014, from <http://stackoverflow.com/questions/2051632/ie8-xss-filter-what-does-it-really-do>
- [14] Naraine, R. “Security gone awry: IE 8 XSS filter exposes sites to XSS attacks”. Retrieved 11 Dec. 2014, from <http://www.zdnet.com/blog/security/security-gone-awry-ie-8-xss-filter-exposes-sites-to-xss-attacks/6221>
- [15] Ross, D. “IE 8 XSS Filter Architecture / Implementation”. Retrieved 11 Dec. 2014, from <http://blogs.technet.com/b/srd/archive/2008/08/19/ie-8-xss-filter-architecture-implementation.aspx>
- [16] “Cloudflare the web performance & security company”. Retrieved 11 Dec. 2014, from <http://www.CloudFlare.com>
- [17] “Cloud computing services and content delivery network (cdn) provider”. Retrieved 11 Dec. 2014, from <http://www.akamai.com>
- [18] Schmitt, I., & Schinzel, S. “WAFFle: fingerprinting filter rules of web application firewalls”. *Proceedings of the 6th USENIX conference on Offensive Technologies*, p. 4, 2012. Retrieved 11 Dec. 2014, from <https://www.usenix.org/system/files/conference/woot12/woot12-final2.pdf>
- [19] “ModSecurity: Open Source Web Application Firewall”. Retrieved 11 Dec. 2014, from <http://modsecurity.org/>
- [20] “Reference Manual . SpiderLabs/ModSecurity Wiki . GitHub”. Retrieved 11 Dec. 2014, from <https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual>
- [21] Canali, D., Balzarotti, D., & Francillon, A. “The role of web hosting providers in detecting compromised websites”. *Proceedings of the 22nd international conference on World Wide Web*, pp. 177–188, 2013.
- [22] Krueger, T., Gehl, C., Rieck, K., & Laskov, P. “TokDoc: a self-healing web application firewall”. *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 1846–1853, 2010.

- [23] “Kubuntu - Friendly Computing”. Retrieved 11 Dec. 2014, from <http://www.kubuntu.org/>
- [24] “Welcome! - The Apache HTTP Server Project”. Retrieved 11 Dec. 2014, from <http://httpd.apache.org/>
- [25] “MySQL :: The world’s most popular open source database”. Retrieved 11 Dec. 2014, from <http://www.mysql.com/>
- [26] “PHP: Hypertext Preprocessor”. Retrieved 11 Dec. 2014, from <http://php.net/>
- [27] “phpMyAdmin”. Retrieved 11 Dec. 2014, from http://www.phpmyadmin.net/home_page/index.php
- [28] “Free Dynamic DNS - Managed DNS - Managed Email - Domain Registration - No-IP”. Retrieved 11 Dec. 2014, from <http://www.noip.com/>
- [29] “Chrome”. Retrieved 11 Dec. 2014, from <http://www.google.com/chrome/>
- [30] “Download Firefox - Free Web Browser - Mozilla”. Retrieved 11 Dec. 2014, from <https://www.mozilla.org/en-US/firefox/new/>
- [31] “phpBB . Free and Open Source Forum Software”. Retrieved 11 Dec. 2014, from <https://www.phpbb.com/>
- [32] “Welcome to Python.org”. Retrieved 11 Dec. 2014, from <http://www.python.org/>
- [33] “mechanize”. Retrieved 11 Dec. 2014, from <http://wwwsearch.sourceforge.net/mechanize>
- [34] “Usage share of web browsers - Wikipedia, the free encyclopedia”. Retrieved 21 Oct. 2014, from http://en.wikipedia.org/wiki/Usage_share_of_web_browsers
- [35] “Use of PERF_RULES variable segaults . Issue #547 . SpiderLabs/ModSecurity . GitHub”. Retrieved 11 Dec. 2014, from <https://github.com/SpiderLabs/ModSecurity/issues/547>