# Global Information Assurance Certification Paper

# Using Web Application Firewall to detect and block common web application attacks

*GIAC GCIA Gold Certification*

Author: Issac Museong Kim, iamissac@gmail.com
Advisor: Antonios Atlasis

Accepted: November 17, 2011

Abstract

A web application firewall is not as common as a network firewall is, but it has been catching our eyes in recent security news, security articles and conferences. Enterprise has been adopting this technology because it enhances web application security significantly. But configuring, implementing and maintaining this new technology is not trivial. To be successful in using it, you must understand application's behavior thoroughly and carefully configure the firewall rules. Also, since commercial versions of this technology are expensive to purchase, implement and maintain, it is recommended to start with an open source product, such as Modsecurity, so you can determine if this solution is appropriate for your budget and environment. This paper will show how to analyze common web attacks by using WAF's detection and logging ability along with Apache server's logging ability. Finally, its effectiveness against some simple and some more advanced web attacks will be examined.

# 1. Introduction

Over the last few years, vulnerabilities in web applications have been the biggest threat in information technology (IT) environment (Modsecurity, 2011). According to the open source vulnerability database (OSVDB), web application threats become almost fifty percent of all vulnerabilities in 2010 (HP DVlabs, 2010). You don't have to be an expert in IT field to figure out that web applications are being used widely in our everyday life as well as in the business sector. Therefore, securing web applications is becoming one of the most important things you need to pay attention as an end-user or as a business user (HP DVlabs, 2010).

A web application firewall (WAF) is a type of firewall that filters HTTP traffic based on a rule set. It inspects the application layer so it usually comes as an appliance type or as a server module. It generally identifies and blocks common web attacks such as cross-site scripting (XSS) and SQL Injection by customizing the rules. Therefore, the customization of its rules is very significant and requires high maintenance (Owasp, 2011).

There are many ways to protect a web application, such as implementing a secure coding practice, managing secure configuration, performing vulnerability assessment and deploying a web application firewall, but there is no silver bullet that it will protect the application entirely. Using a web application firewall is just one method that helps to protect such an application. This technology is relatively new comparing with other technologies, but it can become a powerful solution when you configure and use it properly (Mischel, 2009).

In this paper, Modsecurity is going to be used to demonstrate how to secure a web application using a WAF. Modsecurity protects web applications from a range of web attacks and allows monitoring of HTTP traffic with not many interference in the existing infrastructure (Modsecurity, 2011). It is an open source WAF module for the Apache web server and it has been maintained by SpiderLabs, Trustwave. Since this is an open source product, it comes with free license and many users contribute to the community to improve and maintain the product (Trustwave, 2011).

A WAF is not a tool that just blocks the malicious activity on the application layer. It can also be used to analyze and detect malicious traffic that attacks your critical application. Therefore, this paper will also show how to analyze common web attacks by using WAF's detection and logging ability along with the Apache server's logging ability.

Issac Museong Kim, iamissac@gmail.com

# 2. Setting up and Configuring Modsecurity

## 2.1. System Setup

This section describes the configuration of the testing lab. Figure 1 below shows the Modsecurity architecture in a diagram. First, Fedora Linux 15 is used to host the application. It requires some libraries such as libapr, libapr-util, libpcre, libxml2 and liblua (Modsecurity, 2011). If the Apache server is not installed, the latest version of Apache is recommended. In this lab, the Apache version 2.2.19 is used. After installing the Apache server, the damn vulnerable web application (DVWA) is installed to illustrate the use of a WAF. DVWA is a framework of a deliberately vulnerable web application to help the security testers to learn and test their security skills regarding the protection of a web application (Ivey, 2010). Finally, Modsecurity 2.5.13 is used to wrap the DVWA application in this lab. In order to use Modsecurity, it first needs to be compiled. The compilation creates a file called mod_security2.so which is the Modsecurity plug-in module for the Apache web server. The following line needs to be added to the Apache configuration file http.conf, as shown below, to enable the Modsecurity module:

*LoadModule security2_module modules/mod_security2.so*

After this line has been added, the Apache server needs to be restarted so as the Modsecurity module to run within the Apache web server.

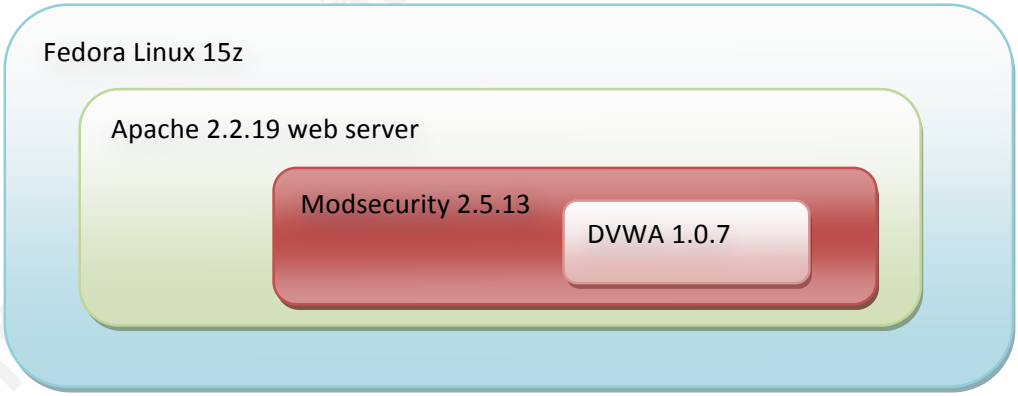


Figure 1: Modsecurity Architecture.

## 2.2. Basic Configuration

Modsecurity has a configuration file called “modsec.conf”. It is recommended to put the file into the directory /etc/httpd/conf.d/ so it gets loaded automatically whenever the Apache

server is restarted. This configuration file contains startup rule sets with auditing settings. The startup configuration is shown below.

```
<IfModule security2_module>
      # Turn rule engine  on and set default action
      SecRuleEngine On
      SecDefaultAction "phase:2,deny,log,status:403"
      #Turn Audit on
      SecAuditEngine On
      SecAuditLog logs/modsec_audit.log
</IfModule>
```

Figure 2: Modsecurity startup configuration file.

It is important to understand what each line means. First of all, all rules and configuration statements must be written between the tags <IfModule> and </IfModule>. Otherwise, it is going to break the configuration or the rule can be ignored. "#" sign is used to put a comment. "SecRuleEngine On" line turns the rule engine on which enables the rule processing. "SecDefault Action" line takes actions when there is a matching rule in place. In this example, it is going to deny the request with 403 HTTP error code and to write the result to the Apache error log and Modsecurity audit log when there is any matching rule. However, since there is no SecRule in the current configuration, it will allow any requests. "SecAuditEngine On" line turns audit logging on for any transactions (Mischel, 2009). "SecAuditLog logs/modsec_audit.log" line enables logs to be written to a modsec_audit.log file. In order to apply the configuration changes, the Apache daemon must be restarted.

## 2.3. Basics of Rules and CRS (Core Rule Set)

Modsecurity needs rules to operate. In Modsecurity, each one of these rules is called SecRule. SecRule has many features and functions. The basic syntax of a rule, which will be briefly explained later, is shown below:

*SecRule Target Operation [Actions]*

The 'Target' variable is a part of the request or the response that is going to be examined. The 'Operation' is a part of the rule that is going to be compared with a matching variable. By default, it uses regular expressions if nothing is specified. The 'Actions' are optional variables

that perform specific actions when there is a matching variable. For example, it can either allow or deny the web traffic by also returning the corresponding status codes. If no actions are specified, it will take the settings from the SecDefaultAction statement (Mischel, 2009). Modsecurity does not provide general protection against normal web attacks by default, unless you install the appropriate rules. Writing such rules can be a complex and a time consuming process. Therefore, Trustwave's SpiderLabs founded the OWASP Modsecurity core rule set (CRS) project. While intrusion detection systems depend on vulnerability signatures, CRS helps Modsecurity to protect web applications from unknown vulnerabilities. The CRS rule set is very easy to follow and deploy since it provides excellent comments; the most updated rule sets can be downloaded from the OWASP Modsecurity CRS Project Site (Modsecurity, 2011).

# 3. Analyzing and blocking common web application attacks

In this section, the most common vulnerabilities such as XSS and SQL injection are exploited against DVWA. First, the attacks run without the Modsecurity blocking rules in place and the results are analyzed by correlating Wireshark traffic captures, Apache access/error logs and Modsecurity audit logs. Then, the appropriate blocking rule is placed in the Modsecurity configuration file based on the analysis. Lastly, the attack runs again to verify the rule by observing the response as well as by analyzing the logs to check if the attack is stopped.

## 3.1. Analyzing XSS attacks

"XSS vulnerability enables an attacker to target other users of the application, potentially gaining access to their data, performing unauthorized actions on their behalf, or carrying out other attacks against them" (Stuttard & Pinto, 2007). In this study, the XSS injection string, "%3C%2FTITLE%3E%3CSCRIPT%3Ealert%28%22XSS%22%29%3B%3C%2FSCRIPT%3E#" is injected into the "name" parameter, as shown below:

*http://127.0.0.1/dvwa/vulnerabilities/xss_r/?name=%3C%2FTITLE%3E%3CSCRIPT%3Ealert%28%22XSS%22%29%3B%3C%2FSCRIPT%3E#*

Then the browser responds with a popup window displaying the message "XSS", as shown in Figure 3.

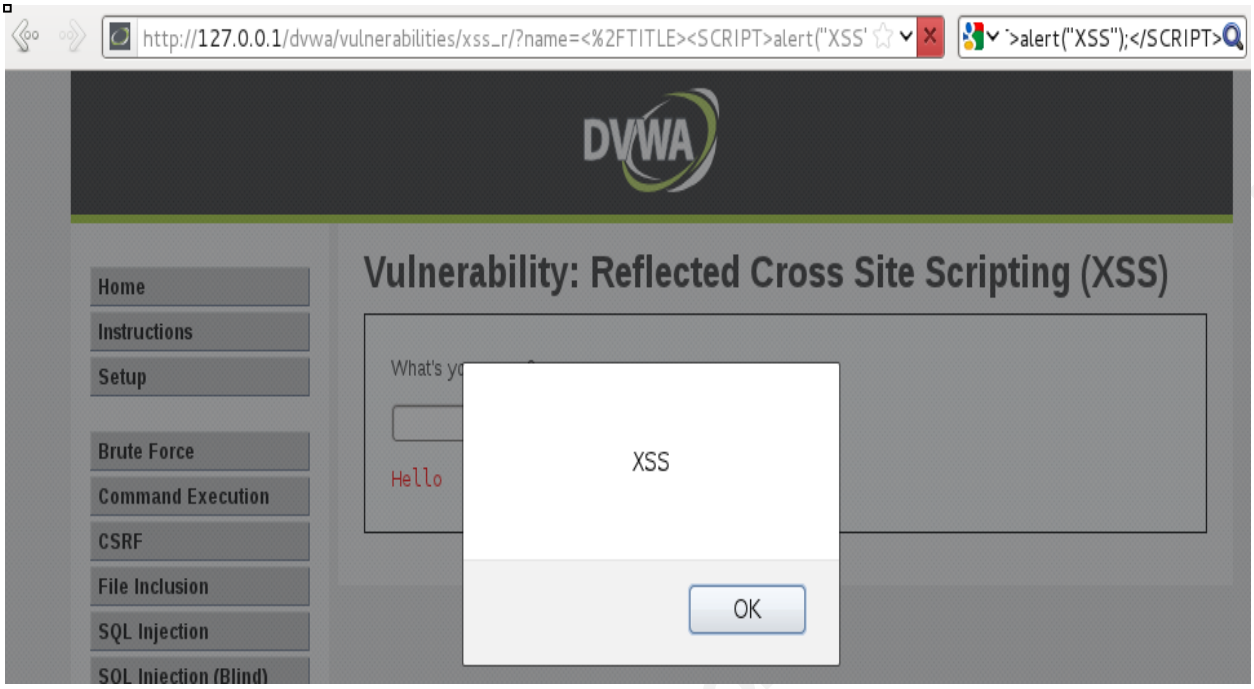


Figure 3: Browser screenshot of the XSS attack without an XSS blocking rule.

Figure 4 shows the Wireshark capture during the attack. It shows a GET request with an XSS injection script in the packet #4. Then, the packet #6 shows that it accepted the request and runs the XSS script.

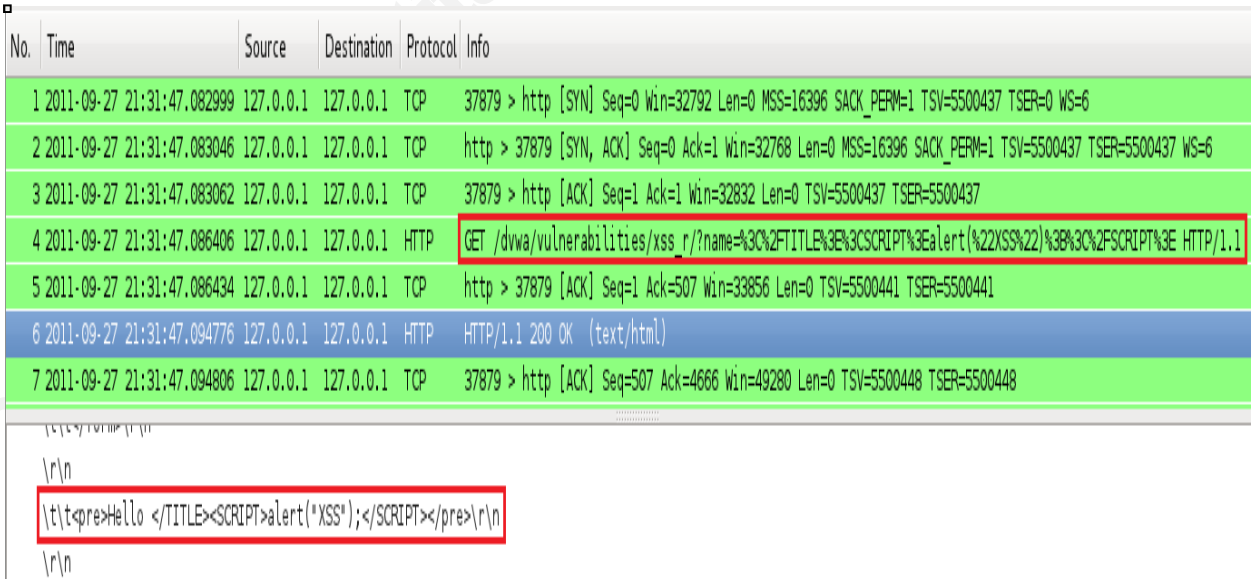


Figure 4: Wireshark screenshot of the XSS attack without an XSS blocking rule.

Issac Museong Kim, iamissac@gmail.com

Figure 5 is from the Apache access log, which is located at "/etc/httpd/logs/access_log" and it shows that it accepted the request to the web server.

```
127.0.0.1 - - [27/Sep/2011:21:31:47 -0700] "GET
/dvwa/vulnerabilities/xss_r/?name=%3C%2FTITLE%3E%3CSCRIPT%3Ealert(%22X
SS%22)%3B%3C%2FSCRIPT%3E HTTP/1.1" 200 4372 "-" "Mozilla/5.0 (X11;
Linux i686; rv:6.0.2) Gecko/20100101 Firefox/6.0.2"
```

Figure 5: Apache access log without an XSS blocking rule.

Figure 6 is from the Modsecurity audit log, which is located at "/etc/httpd/logs/modsec_audit.log. The Modsecurity audit log has certain formats. First, it starts with a unique identifying string such as "cb5e7204" to group the rest of the log entry. After this string, it has a capital letter, such as "A", which represents the audit log part. For example, "A" represents an audit log header, "B" represents a request header, "F" represents a response header, "H" represents an audit log trailer and "Z" represents the end of an audit log entry. There are additional parts you can add by adding "SecAuditLogParts" line, but A, B, F, H and Z are the default settings. As shown in Figure 6, Modsecurity allowed the request because by default, it allows any requests that do not match with any SecRule.

```
--cb5e7204-A--
[27/Sep/2011:21:31:47 --0700] ToKjM38AAAEAAAvEEjAAAAAH 127.0.0.1 37879
127.0.0.1 80
--cb5e7204-B--
GET
/dvwa/vulnerabilities/xss_r/?name=%3C%2FTITLE%3E%3CSCRIPT%3Ealert(%22X
SS%22)%3B%3C%2FSCRIPT%3E HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:6.0.2) Gecko/20100101
Firefox/6.0.2
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

Issac Museong Kim, iamissac@gmail.com

```
Connection: keep-alive
Cookie: security=low; PHPSESSID=ikat12q2sh3l43gnrqoqt4huf4
Cache-Control: max-age=0
--cb5e7204-F--
HTTP/1.1 200 OK
X-Powered-By: PHP/5.3.8
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Content-Length: 4372
Connection: close
Content-Type: text/html;charset=utf-8
--cb5e7204-H--
Apache-Handler: php5-script
Stopwatch: 1317184307086553 8511 (263 266 -)
Producer: ModSecurity for Apache/2.5.13 (http://www.modsecurity.org/).
Server: Apache/2.2.21 (Fedora)
--cb5e7204-Z--
```

Figure 6: Modsecurity Audit Log without an XSS blocking rule.

## 3.2. Blocking XSS attacks

From the various logs in the above section, it is found that this XSS attack uses keywords such as “SCRIPT” and “alert” in the uniform resource identifier (URI). The easy and quick way to block this type of XSS attack is using a Target variable called “REQUEST_URI” which examines a text in URI. For example, the following line is added to the configuration:

*SecRule REQUEST_URI "SCRIPT"|”alert”*

This rule denies any requests that include the words “SCRIPT” or “alert” in their URI. Though this rule does not have a “deny” variable, it denies such requests because the “SecDefaultAction” is set to “deny” in Figure 2. However, this is a very simple rule that can block only simple XSS attacks. An attacker can bypass this type of filtering by encoding or by injecting the script into other places, such as a cookie field. To block more advanced XSS attacks,

you can add common attack strings to the existing rules, such as the ones presented in Table1, or to include the corresponding XSS base rules from the CRS rule set.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Jscript | onsubmit | copyparentfolder | document | javascript | meta | onchange | onmove |
| onerror | onselect | onmouseover | onfocus | javascript: | alert | background | onunload |
| iframe | lowsrc | onmousemove | vbscript | livescript: | script | @import | onresize |

Table1: Common XSS attack strings.

When the same request was sent after placing the rule, the browser responded with a "Forbidden" error code, as shown in Figure 7.


Figure 7: Browser response with the XSS blocking rule

Figure 8 shows the Wireshark capture during the attack. It shows the GET request that it was sent with the aforementioned XSS injection script in the packet #4. Then, the packet #6 shows that it forbids the request.

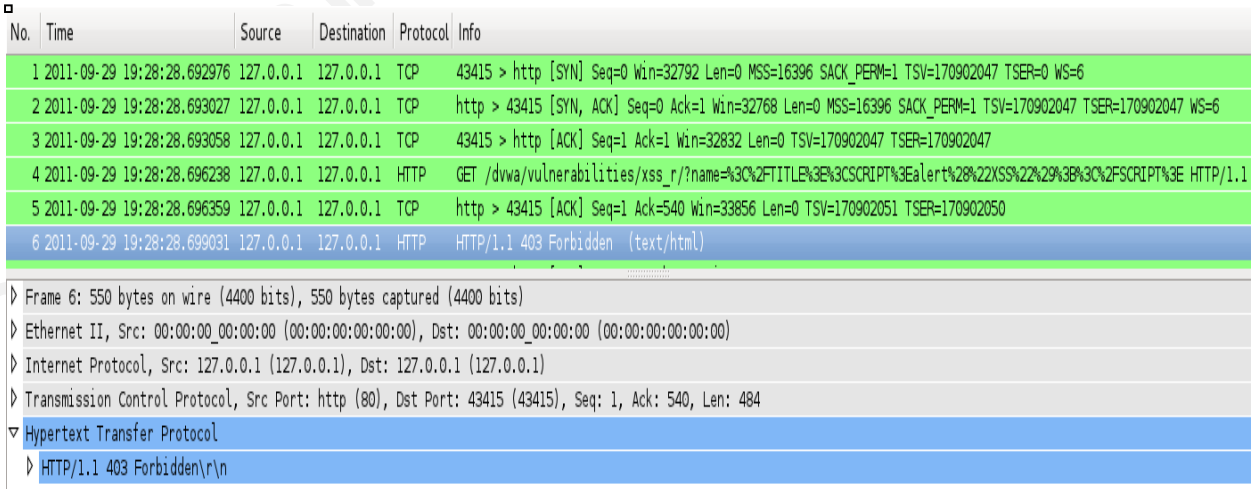


Figure 8: Wireshark screenshot of the XSS attack with the XSS blocking rule.

Issac Museong Kim, iamissac@gmail.com

Figure 9 is from the Apache error log which shows that it denied the request to the web server.

```
 [Thu Sep 29 19:28:28 2011] [error] [client 127.0.0.1] ModSecurity:
Access denied with code 403 (phase 2). Pattern match "(SCRIPT|alert)"
at REQUEST_URI. [file "/etc/httpd/conf.d/modsec.conf"] [line "15"]
[hostname "127.0.0.1"] [uri "/dvwa/vulnerabilities/xss_r/"] [unique_id
ToUpTH8AAAEAAEgDDEYAAAAB"]
```

Figure 9: Apache error log with the XSS blocking rule.

As shown in Figure 10, the Modsecurity denied the request because it found the pattern in the rule.

```
--22a47b2f-A--
[29/Sep/2011:19:28:28 --0700] ToUpTH8AAAEAAEgDDEYAAAAB 127.0.0.1 43415
127.0.0.1 80
--22a47b2f-B—
GET
/dvwa/vulnerabilities/xss_r/?name=%3C%2FTITLE%3E%3CSCRIPT%3Ealert%28%2
2XSS%22%29%3B%3C%2FSCRIPT%3E HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:6.0.2) Gecko/20100101
Firefox/6.0.2
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://127.0.0.1/dvwa/vulnerabilities/xss_r/
Cookie: security=low; PHPSESSID=ikat12q2sh3l43gnrqoqt4huf4
--22a47b2f-F—
HTTP/1.1 403 Forbidden
Content-Length: 304
Connection: close
```

Issac Museong Kim, iamissac@gmail.com

```
Content-Type: text/html; charset=iso-8859-1
--22a47b2f-H–
Message: Access denied with code 403 (phase 2). Pattern match
"(SCRIPT|alert)" at REQUEST_URI. [file
"/etc/httpd/conf.d/modsec.conf"] [line "15"]
Action: Intercepted (phase 2)
Stopwatch: 1317349708697236 5950 (1302 1605 -)
Producer: ModSecurity for Apache/2.5.13 (http://www.modsecurity.org/).
Server: Apache/2.2.21 (Fedora)
--22a47b2f-Z--
```

Figure 10: Modsecurity Audit Log with the XSS blocking rule.

## 3.3. Analyzing SQL Injection Attacks

SQL Injection is another common web application attack method. This vulnerability allows attackers to inject malicious SQL statements to interact with the backend database. From this injection, the attacker may be able to obtain data as well as to execute malicious commands to the database (Stuttard & Pinto, 2007). In this study, the SQL injection string (' union all select user, password from dvwa.users#) was injected into the "id" parameter, as shown below:

http://127.0.0.1/dvwa/vulnerabilities/sqli/?id=%27+union+all+select+user%2C+password+from+dvwa.users%23&Submit=Submit#

Then, the browser responded with a list of users and the corresponding password hashes, as shown in Figure 11.

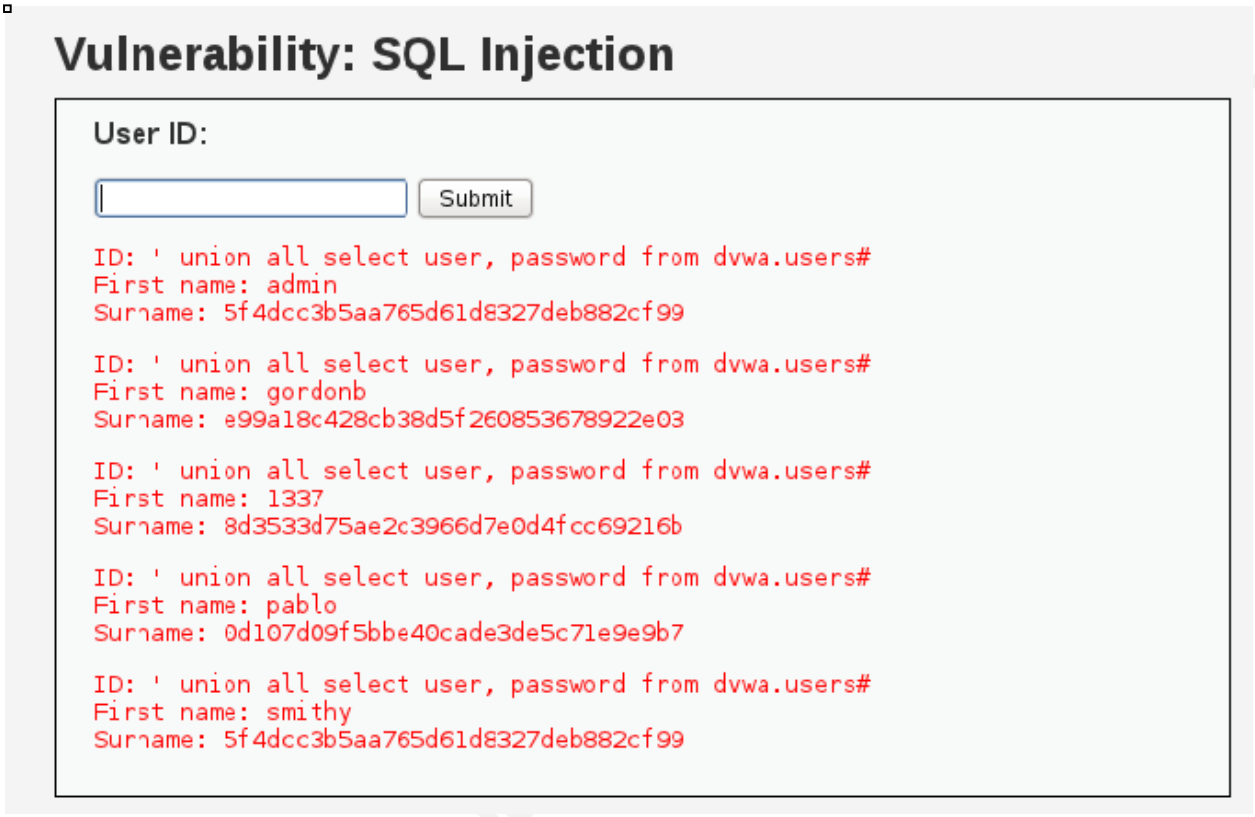


Figure 11: Browser screenshot of the SQL Injection attack without a blocking rule.

Figure 12 shows the Wireshark capture during the attack. It shows a GET request which was sent with an SQL Injection script in the packet #4. Then, the packet #6 shows that it accepted the request and displays the user ID as well as the hash.

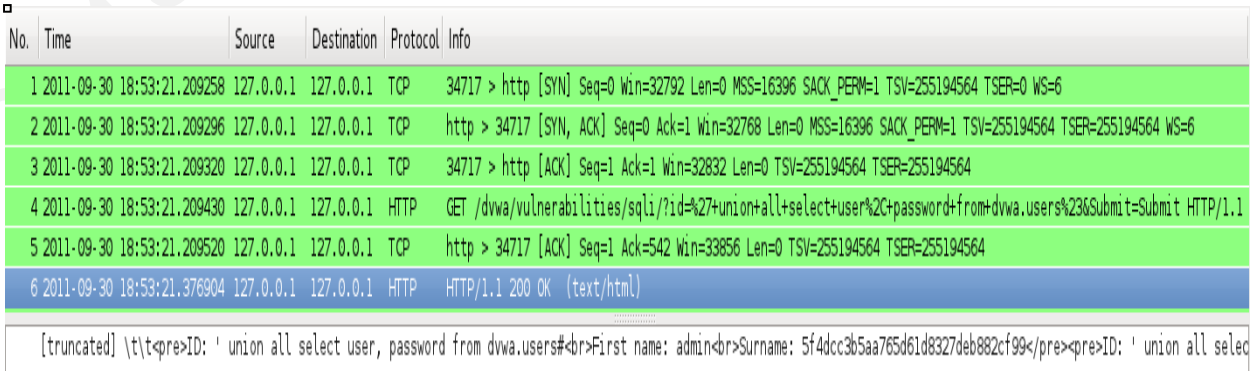


Figure 12: Wireshark screenshot of SQL Injection attack without a blocking rule.

Issac Museong Kim, iamissac@gmail.com

Figure 13 displays the Apache access log, which shows that it accepted the request to the web server.

```
127.0.0.1 - - [30/Sep/2011:18:53:21 -0700] "GET
/dvwa/vulnerabilities/sqli/?id=%27+union+all+select+user%
2C+password+from+dvwa.users%23&Submit=Submit HTTP/1.1" 200 4990
"http://127.0.0.1/dvwa/vulnerabilities/sqli/" "Mozilla/5.0 (X11; Linux
i686; rv:6.0.2) Gecko/20100101 Firefox/6.0.2"
```

Figure 13: Apache access log without an SQL Injection blocking rule.

As shown in Figure 14, the Modsecurity allowed the request because it did not find any matching SecRule.

```
--a396eb61-A--
[30/Sep/2011:18:53:21 --0700] ToZykX8AAAEAAEgEC-oAAAAC 127.0.0.1 34717
127.0.0.1 80
--a396eb61-B--
GET
/dvwa/vulnerabilities/sqli/?id=%27+union+all+select+user%2C+password+f
rom+dvwa.users%23&Submit=Submit HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:6.0.2) Gecko/20100101
Firefox/6.0.2
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://127.0.0.1/dvwa/vulnerabilities/sqli/
Cookie: security=low; PHPSESSID=ikat12q2sh3l43gnrqoqt4huf4
--a396eb61-F--
HTTP/1.1 200 OK
```

Issac Museong Kim, iamissac@gmail.com

```
X-Powered-By: PHP/5.3.8
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Content-Length: 4990
Connection: close
Content-Type: text/html;charset=utf-8
--a396eb61-H--
Apache-Handler: php5-script
Stopwatch: 1317434001221137 169753 (20956 20986 -)
Producer: ModSecurity for Apache/2.5.13 (http://www.modsecurity.org/).
Server: Apache/2.2.21 (Fedora)
--a396eb61-Z--
```

Figure 14: Modsecurity Audit Log without an SQL Injection blocking rule.

## 3.4. Blocking SQL Injection attack

From the various logs in the above section, it is found that this SQL Injection attack uses a keyword "union" in the argument. Similarly to the prevention of the XSS attack, you can add the following SecRule line to the configuration to block the SQL Injection attack. In this example, "msg" argument is added to the rule so it writes the log message that explains why it is blocked:

*SecRule ARGS "union" "msg: 'SQL Injection'"*

This rule denies any requests that include the keyword "union" as an argument. Again this is a very simple type of SQL injection attacks. To block more advanced SQL Injection attacks, you can add more common attack strings to the existing rule, such as the ones presented in Table 2, or the corresponding SQL Injection base rules from the CRS rule set should be included.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| xor | rlike | -- | and | # | union | All | ; |
| ' | drop | delete | having | 1=1 | admin | Select | |

Table 2: Common SQL Injection attack strings.

When the same request was sent after placing the rule, the browser responded with "Forbidden" error code, as shown in Figure 15.


Figure 15: Browser response with the SQL Injection blocking rule.

Figure 16 shows the Wireshark capture during the attack. It shows the GET request which was sent with the aforementioned SQL injection script in the packet #4. Then, the packet #6 shows that it forbids the request.

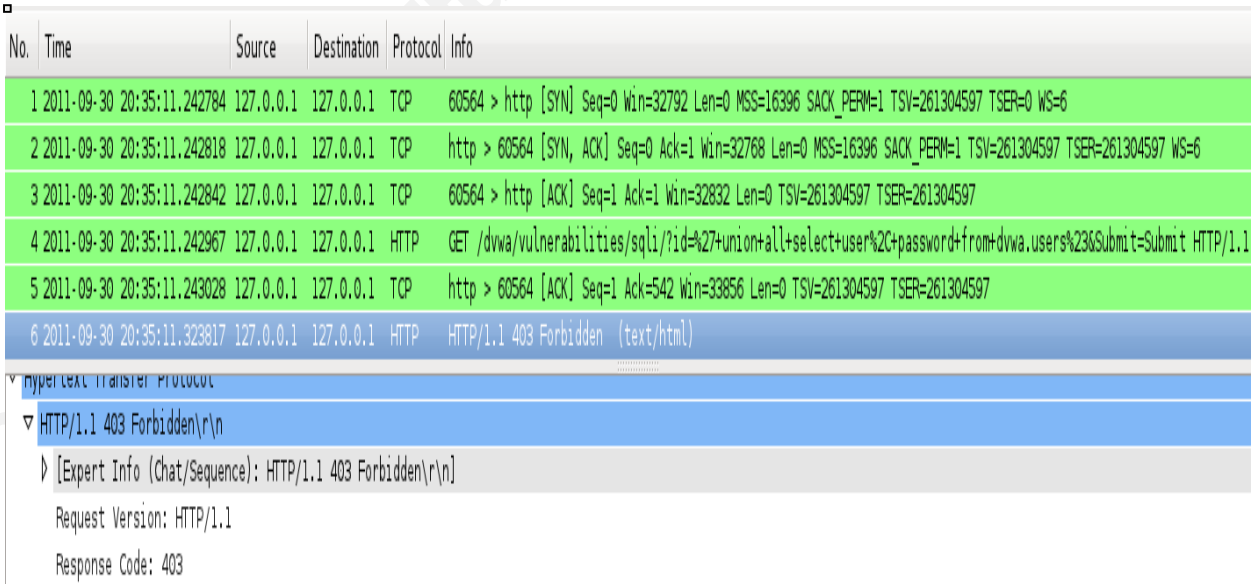


Figure 16: Wireshark screenshot of the SQL Injection attack with the SQL Injection blocking rule.

Issac Museong Kim, iamissac@gmail.com

Figure 17 displays the Apache error log that shows it denied the request to the web server.

```
[Fri Sep 30 20:35:11 2011] [error] [client 127.0.0.1] ModSecurity:
Access denied with code 403 (phase 2).
 Pattern match "union" at ARGS:id. [file
"/etc/httpd/conf.d/modsec.conf"] [line "16"] [msg "SQL Injection"]
[hostname "127.0.0.1"] [uri "/dvwa/vulnerabilities/sqli/"] [unique_id
"ToaKb38AAAEAAGXcEtcAAAAG"]
```

Figure 17: Apache error log with the SQL Injection blocking rule.

As shown in Figure 18, the Modsecurity denied the request, because it found the pattern in the rule.

```
--2ef14200-A--
[30/Sep/2011:20:35:11 --0700] ToaKb38AAAEAAGXcEtcAAAAG 127.0.0.1 60564
     127.0.0.1 80
--2ef14200-B--
GET
     /dvwa/vulnerabilities/sqli/?id=%27+union+all+select+user%2C+passw
     ord+from+dvwa.users%23&Submit=Submit HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:6.0.2) Gecko/20100101
     Firefox/6.0.2
Accept:
     text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Referer: http://127.0.0.1/dvwa/vulnerabilities/sqli/
Cookie: security=low; PHPSESSID=ikat12q2sh3l43gnrqoqt4huf4
--2ef14200-F--
HTTP/1.1 403 Forbidden
Content-Length: 303
Connection: close
Content-Type: text/html; charset=iso-8859-1
```

```
--2ef14200-H--
Message: Access denied with code 403 (phase 2). Pattern match "union"
     at ARGS:id. [file "/etc/httpd/conf.
d/modsec.conf"] [line "16"] [msg "SQL Injection"]
Action: Intercepted (phase 2)
Stopwatch: 1317440111243660 80417 (793 79882 -)
Producer: ModSecurity for Apache/2.5.13 (http://www.modsecurity.org/).
Server: Apache/2.2.21 (Fedora)
--2ef14200-Z--
```

Figure 18: Modsecurity Audit Log with the SQL Injection blocking rule.

# 4. Attacking the application using automated tools

In this section, automated testing tools such as XSS Me and Sqlmap are used to test for XSS and SQL injection vulnerabilities. These automatic testing tools perform more complex and a wider range of attacks to make sure the SecRule is safe enough to block such attacks. The use of these tools will be explained briefly. Also, when the existing rules are not good enough to block an attack, more comprehensive rules will be added after the appropriate analysis.

## 4.1. XSS attacks using XSS Me

XSS Me is a Firefox plug-in created by Security Compass. This tool helps testers to run XSS attacks against a target website instantly while browsing (Security Compass, 2010). XSS Me performs 154 different types of XSS test by default. To make the test more comprehensive, the attack strings from XSS cheat sheet by RSnake were added to the existing strings of XSS Me (Hansen, 2008). So, 328 XSS tests were totally performed against the web application. To run XSS Me, you first have to get to the URL that needs to be tested and then open the XSS Me side bar. Figure 19 shows the interface of XSS Me. The “name” parameter was selected, then “run all tests” was selected to run the attack. XSS Me runs against the same URL that was tested in section 3.1 with the existing SecRule.

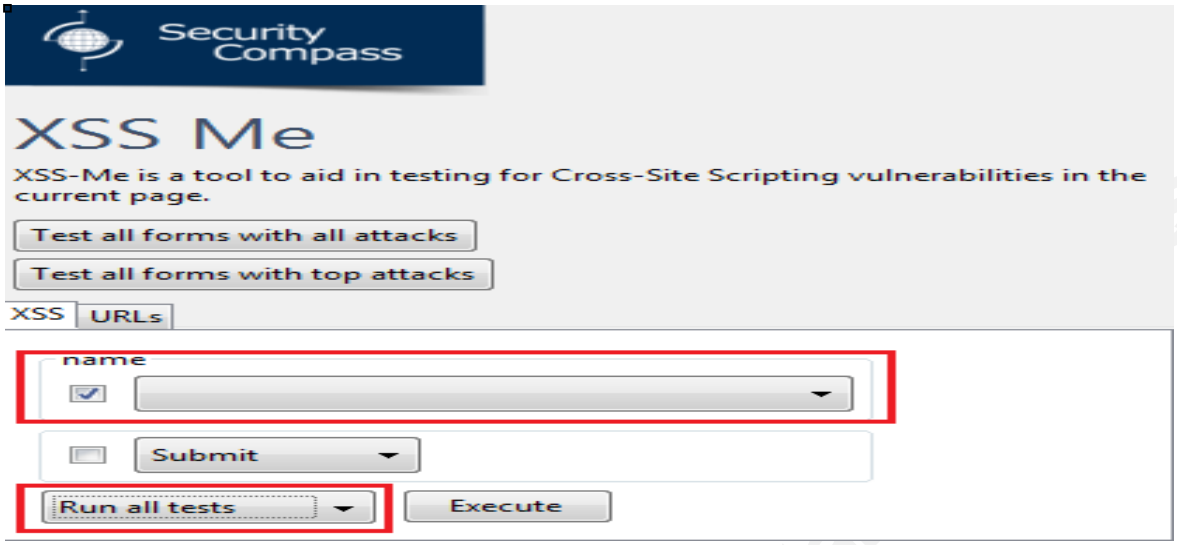


Figure 19: XSS Me interface.

Figure 20 shows that there were 6 Failures meaning that 6 XSS tests were successful while there are 162 Warnings meaning that 162 XSS tests were not successful but it may were vulnerable in other environment, and 160 XSS tests were passed meaning that 160 XSS tests were not successful.

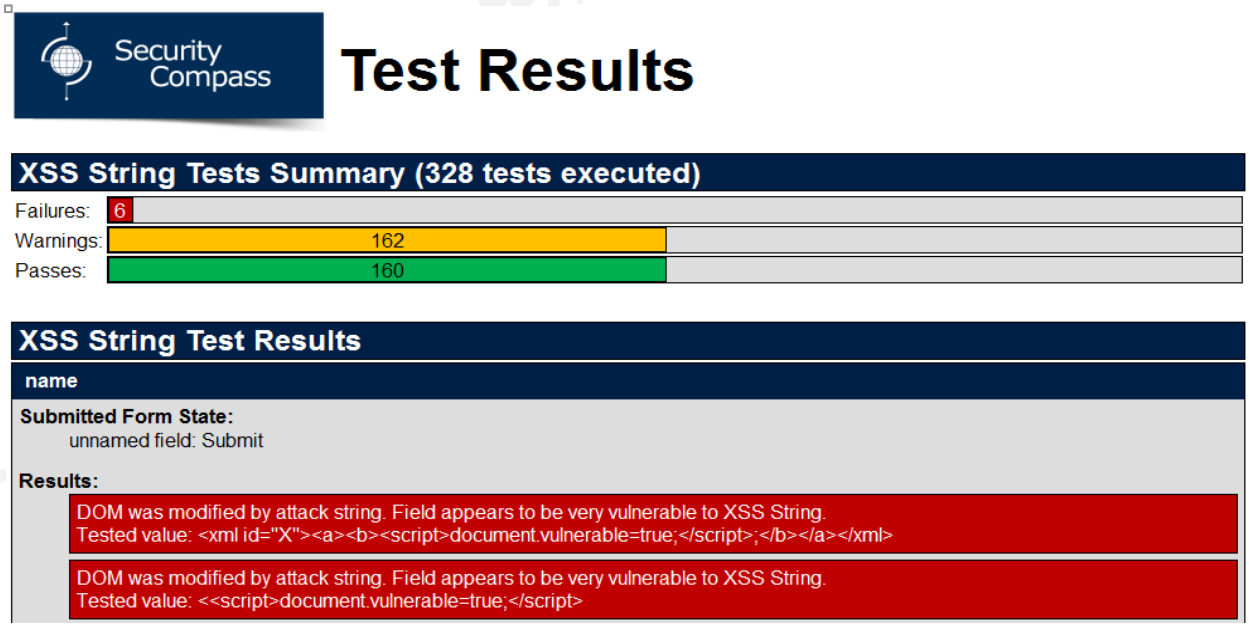


Figure 20: XSS testing with the existing SecRule using the XSS Me tool

Issac Museong Kim, iamissac@gmail.com

To block these 6 XSS attacks, the strings in Table1 are added to the existing SecRule that was used to block the XSS attack, as shown below:

*SecRule REQUEST_URI*

*"(Jscript|onsubmit|copyparentfolder|document|javascript|meta|Onchange|onmove|onerror|onselect|onmouseover|onfocus|javascript:|alert|background|onunload|iframe|lowsrc|onmousemove|vbscript|livescript:|script|@import|onresize)"*

*SecRule ARGS*

*"(Jscript|onsubmit|copyparentfolder|document|javascript|meta|Onchange|onmove|onerror|onselect|onmouseover|onfocus|javascript:|alert|background|onunload|iframe|lowsrc|onmousemove|vbscript|livescript:|script|@import|onresize)"*

Figure 21 shows the test results with the updated SecRule in place. Out of 328 tests, there were 0 failures meaning that all XSS tests were not successful and that they were blocked by Modsecurity.

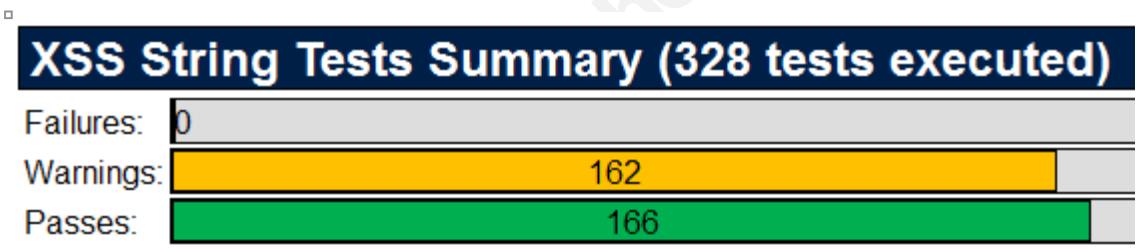


Figure 21: XSS testing with the updated SecRule using the XSS Me tool.

Lastly, the application was tested by using some of the XSS obfuscated strings that were presented in the Black Hat USA 2009 by Eduardo Vela and David Linsay, as shown below (Vela & Lindsay, 2009):

*<img src="x:alert" onerror="eval(src%2b'(0)')">*

*<img src="x:gif" onerror="eval('al'%2b'lert(0)')">*

*<img src="x:gif" onerror="window['al\u0065rt'](0)"></img>*

Figure 22 from the Modsecurity audit logs shows that these obfuscated attacks are blocked by Modsecurity. However, this does not mean that the web application is safe against any future obfuscated attack; hence, it is recommended to include the CRS rule set and update them periodically.

Issac Museong Kim, iamissac@gmail.com

```
GET
/dvwa/vulnerabilities/xss_r/?name=%3Cimg+src%3D%22x%3Aalert%22+onerro
r%3D%22eval%28src%252b%27%280%29%27%29%22%3E HTTP/1.1
--cf6ca670-H--
Message: Access denied with code 403 (phase 2). Pattern match
"(Jscript|onsubmit|copyparentfolder|document|javascript|meta|On
change|onmove|onerror|onselect|onmouseover|onfocus|javascript:|alert|
background|onunload|iframe|lowsrc|onmousemove|vbscript|livescript:|sc
ript|@import|onresize)" at REQUEST_URI. [file
"/etc/httpd/conf.d/modsec.conf"] [line "20"] [msg "XSS attack"]
--cf6ca670-B--
GET
/dvwa/vulnerabilities/xss_r/?name=%3Cimg+src%3D%22x%3Agif%22+onerror%
3D%22eval%28%27al%27%252b%27lert%280%29%27%29%22%3E  HTTP/1.1
--cf6ca670-H--
Message: Access denied with code 403 (phase 2). Pattern match
"(Jscript|onsubmit|copyparentfolder|document|javascript|meta|On
change|onmove|onerror|onselect|onmouseover|onfocus|javascript:|alert|
background|onunload|iframe|lowsrc|onmousemove|vbscript|livescript:|sc
ript|@import|onresize)" at REQUEST_URI. [file
"/etc/httpd/conf.d/modsec.conf"] [line "20"] [msg "XSS attack"]
--cf6ca670-B--
GET
/dvwa/vulnerabilities/xss_r/?name=%3Cimg+src%3D%22x%3Agif%22+onerror%
3D%22window%5B%27al%5Cu0065rt%27%5D+%280%29%22%3E%3C%2Fimg%3E
HTTP/1.1
--cf6ca670-H--
Message: Access denied with code 403 (phase 2). Pattern match
"(Jscript|onsubmit|copyparentfolder|document|javascript|meta|On
change|onmove|onerror|onselect|onmouseover|onfocus|javascript:|alert|
background|onunload|iframe|lowsrc|onmousemove|vbscript|livescript:|sc
ript|@import|onresize)" at REQUEST_URI. [file
"/etc/httpd/conf.d/modsec.conf"] [line "20"] [msg "XSS attack"]
```

Figure 22: Modsecurity audit log of obfuscated XSS attacks with the updated SecRule

## 4.2. SQL injection attacks using Sqlmap

The Sqlmap is "an open source penetration testing tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers" (Damele & Stampar 2011). Python interpreter version 2 or newer is required to run this tool. In this study, the following command was run to identify SQL injection vulnerabilities:

*./sqlmap.py -u 'http://127.0.0.1/dvwa/vulnerabilities/sqli/index.php?id=1&Submit=Submit#' --cookie='security=low; PHPSESSID=ikat12q2sh3l43gnrqoqt4huf4' --dbms=MySQL --tamper='tamper/randomcase.py, tamper/charencode.py' --level=5 --risk=3*

Understanding the command above is important. First, "-u" switch sets the target URL and "-- cookie" sets the cookie value. DVWA application uses a cookie value to authenticate the users, so this value is required whenever the request is sent to the application. "--dbms" sets the type of the database of the target application so it only injects the attack strings that works for the specific database. In this experiment, it is already known that it uses a MySQL database from the DVWA documentation (Ivey, 2011). "--tamper" option allows the obfuscation of the attack strings in order to bypass the detection. "randomcase.py" script randomizes the attack strings and "charencode.py" script encodes the characters. "--level" option sets the level of test to perform meaning that level 1 uses limited number of tests whereas level 5 uses much larger amount of tests. "--risk" option sets the risk of the tests to perform, meaning that risk 1 uses a limited number of injection points whereas risk 3 uses a much larger amount of injection points (Damele & Stampar 2011). Figure 23 shows the output of the testing against the SecRule that was created in Section 3.4. The highlighted lines show that the tool injected 211 tests and the parameter "id" is vulnerable.

```
sqlmap/1.0-dev (r4397) - automatic SQL injection and database takeover
tool
[18:10:54] [INFO] loading tamper script 'randomcase'
[18:10:54] [INFO] loading tamper script 'charencode'
 [18:10:54] [INFO] testing connection to the target url
[18:10:55] [INFO] testing if the url is stable, wait a few seconds
[18:10:56] [INFO] url is stable
[18:10:56] [INFO] testing if GET parameter 'id' is dynamic
```

```
[18:10:56] [WARNING] GET parameter 'id' appears to be not dynamic
[18:10:56] [INFO] heuristic test shows that GET parameter 'id' might
be injectable (possible DBMS: MySQL)
[18:10:56] [INFO] testing sql injection on GET parameter 'id'
[18:10:56] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause'
[18:10:58] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause (Generic comment)'
[18:10:59] [INFO] testing 'OR boolean-based blind - WHERE or HAVING
clause'
[18:11:00] [INFO] GET parameter 'id' is 'OR boolean-based blind -
WHERE or HAVING clause' injectable
[18:11:00] [INFO] testing 'MySQL >= 5.0 AND error-based - WHERE or
HAVING clause'
[18:11:00] [INFO] GET parameter 'id' is 'MySQL >= 5.0 AND error-based
- WHERE or HAVING clause' injectable
[18:11:00] [INFO] testing 'MySQL > 5.0.11 stacked queries'
[18:11:00] [INFO] testing 'MySQL < 5.0.12 stacked queries (heavy
query)'
[18:11:00] [INFO] testing 'MySQL > 5.0.11 AND time-based blind'
[18:11:10] [INFO] GET parameter 'id' is 'MySQL > 5.0.11 AND time-based
blind' injectable
[18:11:10] [INFO] testing 'MySQL UNION query (NULL) - 1 to 10 columns'
[18:11:10] [INFO] target url appears to be UNION injectable with 2
columns
[18:11:10] [INFO] GET parameter 'id' is 'MySQL UNION query (NULL) - 1
to 10 columns' injectable
[18:11:10] [WARNING] in OR boolean-based injections, please consider
usage of switch --drop-set-cookie if you experience any problems
during data retrieval
GET parameter 'id' is vulnerable. Do you want to keep testing the
others? [y/N] N
sqlmap identified the following injection points with a total of 211
HTTP(s) requests:
```

Issac Museong Kim, iamissac@gmail.com

```
---
Place: GET
Parameter: id
Type: boolean-based blind
Title: OR boolean-based blind - WHERE or HAVING clause
Payload: id=-2213' OR NOT (7100=7100) AND 'VBXM'='VBXM&Submit=Submit

Type: error-based
Title: MySQL >= 5.0 AND error-based - WHERE or HAVING clause
Payload: id=1' AND (SELECT 3198 FROM(SELECT
COUNT(*),CONCAT(CHAR(58,100,104,110,58),(SELECT (CASE WHEN (3198=3198)
THEN 1 ELSE 0 END)),CHAR(58,103,120,117,58),FLOOR(RAND(0)*2))x FROM
INFORMATION_SCHEMA.CHARACTER_SETS GROUP BY x)a) AND
'Ezsb'='Ezsb&Submit=Submit

Type: UNION query
Title: MySQL UNION query (NULL) - 2 columns
Payload: id=1' UNION ALL SELECT
CONCAT(CHAR(58,100,104,110,58),IFNULL(CAST(CHAR(121,104,70,108,121,115
,88,113,67,112) AS CHAR),CHAR(32)),CHAR(58,103,120,117,58)), NULL# AND
'diwU'='diwU&Submit=Submit

Type: AND/OR time-based blind
Title: MySQL > 5.0.11 AND time-based blind
Payload: id=1' AND SLEEP(5) AND 'AFme'='AFme&Submit=Submit
---
[18:11:18] [INFO] changes made by tampering scripts are not included
in shown payload content(s)
[18:11:18] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Fedora
web application technology: Apache 2.2.21, PHP 5.3.8
back-end DBMS: MySQL 5.0
```

Figure 23: SQL Injection testing with the existing SecRule using the Sqlmap tool.

Issac Museong Kim, iamissac@gmail.com

To block this type of SQL injection attack, the strings in Table2 are added to the existing SecRule, as shown below:

*SecRule ARGS "(union|xor|rlike|--|#|union all|;|'|drop|delete|having|1=1|admin|select|and)" "msg: 'SQL Injection'"*

Figure 24 shows the test results with the SecRule above in place. The highlighted line shows that the SQL injection is not successful and the request was forbidden 8425 times. Even though the attack strings are obfuscated by using the "--tamper" option, it is still blocked by Modsecurity. However, this does not mean that it is safe against any future obfuscated attacks, so it is recommended to include the CRS rule set and update them periodically.

```
[21:19:50] [INFO] loading tamper script 'randomcase'
[21:19:50] [INFO] loading tamper script 'charencode'
[21:19:50] [INFO] using '/home/sqlmap/output/127.0.0.1/session' as
session file
[21:19:50] [INFO] testing connection to the target url
[21:19:50] [INFO] testing if the provided string is within the target
URL page content
[21:19:50] [INFO] testing if GET parameter 'id' is dynamic
[21:19:50] [INFO] confirming that GET parameter 'id' is dynamic
[21:19:50] [INFO] GET parameter 'id' is dynamic
[21:19:50] [WARNING] heuristic test shows that GET parameter 'id'
might not be injectable
[21:19:50] [INFO] testing sql injection on GET parameter 'id'
[21:21:15] [WARNING] GET parameter 'id' is not injectable
[21:21:15] [INFO] testing if GET parameter 'Submit' is dynamic
[21:21:15] [WARNING] GET parameter 'Submit' appears to be not dynamic
[21:21:15] [WARNING] heuristic test shows that GET parameter 'Submit'
might not be injectable
[21:21:15] [INFO] testing sql injection on GET parameter 'Submit'
[21:21:15] [INFO] testing 'AND boolean-based blind - WHERE or HAVING
clause'
[21:21:15] [INFO] GET parameter 'Submit' is 'AND boolean-based blind -
WHERE or HAVING clause' injectable
[21:21:21] [WARNING] GET parameter 'Submit' is not injectable
```

Issac Museong Kim, iamissac@gmail.com

```
[21:21:21] [INFO] testing if Referer parameter 'Referer' is dynamic
[21:21:21] [WARNING] Referer parameter 'Referer' appears to be not
dynamic
[21:21:21] [WARNING] heuristic test shows that Referer parameter
'Referer' might not be injectable
[21:21:21] [INFO] testing sql injection on Referer parameter 'Referer'
[21:23:59] [WARNING] Referer parameter 'Referer' is not injectable
[21:23:59] [INFO] testing if User-Agent parameter 'User-Agent' is
dynamic
[21:26:48] [WARNING] User-Agent parameter 'User-Agent' is not
injectable
[21:26:48] [INFO] testing if Cookie parameter 'security' is dynamic
[21:26:48] [WARNING] Cookie parameter 'security' appears to be not
dynamic
[21:26:48] [WARNING] heuristic test shows that Cookie parameter
'security' might not be injectable
[21:26:48] [INFO] testing sql injection on Cookie parameter 'security'
[21:29:53] [WARNING] Cookie parameter 'security' is not injectable
[21:29:53] [INFO] testing if Cookie parameter 'PHPSESSID' is dynamic
sqlmap got a 302 redirect to 'http://127.0.0.1:80/dvwa/login.php'. Do
you want to follow redirects from now on (or stay on the original
page)? [Y/n] n
[21:30:03] [INFO] confirming that Cookie parameter 'PHPSESSID' is
dynamic
[21:30:03] [INFO] Cookie parameter 'PHPSESSID' is dynamic
[21:30:03] [WARNING] heuristic test shows that Cookie parameter
'PHPSESSID' might not be injectable
[21:30:03] [INFO] testing sql injection on Cookie parameter
'PHPSESSID'
[21:35:05] [WARNING] Cookie parameter 'PHPSESSID' is not injectable
[21:35:05] [CRITICAL] all parameters appear to be not injectable.
[21:35:05] [WARNING] HTTP error codes detected during testing:
403 (Forbidden) - 8425 times
```

Figure 24: SQL Injection testing with the updated SecRule using the Sqlmap tool.

Issac Museong Kim, iamissac@gmail.com

Lastly, the application was tested with one of the latest obfuscated SQL injection techniques that were presented by the CWH Underground team (Phongthiproek, 2011). This technique uses a MySQL server's comment feature which is the "/*" character sequence to the following "*/" character sequence with a new line character such as "%0D%0A". Combining these two methods, it allows the attack strings to extend over multiple lines so Modsecurity fails to detect the pattern (Phongthiproek, 2011). The attack string "' union all select user, password from dvwa.users#" from the section 3.3 was obfuscated as following:

*'union%23foo%2F*bar%0D%0Aall%20select%23foo%0D%0Auser%2Cpassword%20from%20from%20dvwa.users%23*

The above stings are equivalent to the following SQL payload:

*' union#foo*/*bar*
*all select#foo*
*user, password from dvwa.users#*

But when these strings are passed to the MySQL database, they are interpreted as following:

*' union all select user, password from dvwa.users#*

Figure 25 from the Modsecurity audit log shows that the attack was not successful and that it was blocked by Modsecurity. However, this may work against different types of SecRule or applications so it is worth to try it. The CWH Underground team proved that the attack was successful against the application that utilizes the CRS SQL injection rule version 2.2.1. As of this writing, the latest version of CRS SQL injection rule is 2.2.2 so, it is recommended to update the rules frequently if the CRS rule set is used.

```
--aa2d3934-A--
[29/Oct/2011:18:01:14 --0700] Tqyh2n8AAAEAAAjUCgoAAAAD 192.168.63.1
     46195 192.16
8.63.157 80
--aa2d3934-B--
GET
     /dvwa/vulnerabilities/sqli/?id=%E2%80%98union%2523foo%252F*bar%25
     0D%250Aall%2520select%2523foo%250D%250Auser%252Cpassword%2520from
     %2520from%2520dvwa.users%2523&Submit=Submit HTTP/1.1
Host: 192.168.63.157
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1)
```

Issac Museong Kim, iamissac@gmail.com

```
Accept:
     text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 115
Connection: keep-alive
Referer: http://192.168.63.157/dvwa/vulnerabilities/sqli/
Cookie: security=low; PHPSESSID=osqeourqqjacdph85t9oi58ji3
--aa2d3934-F--
HTTP/1.1 403 Forbidden
Content-Length: 308
Connection: close
Content-Type: text/html; charset=iso-8859-1
--aa2d3934-H--
Message: Access denied with code 403 (phase 2). Pattern match
     "(union|xor|or|rli
ke|--|#|all|'|drop|delete|having|select|and)" at ARGS:id. [file
     "/etc/httpd/conf
.d/modsec.conf"] [line "24"]
Action: Intercepted (phase 2)
Stopwatch: 1319936474504182 9305 (1552 8952 -)
Producer: ModSecurity for Apache/2.5.13 (http://www.modsecurity.org/).
Server: Apache/2.2.21 (Fedora)
--aa2d3934-Z--
```

Figure 25: Example of an advanced obfuscating SQL Injection attack.

## 5. Successful implementation of Modsecurity

In the above section, the deny rule was added to block the attacks right after the corresponding analysis, but in real world this is a very dangerous approach, because, it may break some legitimate application functions. For example, one of the demonstrated XSS attacks was blocked by restricting the use of the “SCRIPT” string in the URI. This method worked in the

above example, but if a critical function of the application has the word "script" in the URI, this will break the application. As a security professional, it is important to consider availability as well as confidentiality and integrity of the application server. In this section, it is going to be discussed how to implement Modsecurity successfully for your application.

## 5.1. Whitelisting model vs. Blacklisting model

When implementing a WAF for an application, it is very important to choose either the whitelisting or the blacklisting model. With the whitelisting model, exact actions of the application needs to be defined in the rule set and the WAF will only allow the requests that were specified in the rule set. Any other requests will be denied. Therefore, the whitelisting method provides high security and protection against new types of attacks, but it is really difficult to implement because you need a detailed knowledge of the application; any misinterpretation of the application behavior causes the failure of the application (Mischel, 2009).

On the other hand, with the blacklisting model, you only have to specify the requests you want to block and all other request will be allowed. So, it is much easier to implement it than the whitelisting approach and it is less likely that the application will fail. However, it is weak against attacks that were not defined in the rule set.

## 5.2. Log Mode

In the previous section, both the whitelisting and the blacklisting model were discussed, but the last one will be examined further for the rest of the section. In the above testing, the "SecDefaultAction" was set to a deny mode so it only denies every request that is matched by SecRule, while any other request is allowed. Again, this is a very dangerous approach to start with; instead, it is recommended to use the log mode which allows the requests that are matched by a SecRule to create a log entry. To enable the log mode, you need to change the "deny" variable to "pass" from the "SetDefaultAction" line of configuration file, as shown below. If you have used any deny action for individual rules which supersede the default action, you need to change it to "pass, log" as well.

*SecDefaultAction "phase: 2, pass, log"*

When you use the log mode, it does not affect the functionality of the web application because it does not block any request. You need to carefully examine the logs and find out what requests have been logged by the Modsecurity audit log. If a valid request is caught by any

SecRule and appears in the log, you need to modify your SecRule accordingly. But going through the tons of log entries is a difficult task. To help with log auditing, it is a good idea to use the "msg" option, as shown in Section 3.4. Using this "msg" option, you can specify why the traffic is logged and you can find out easily which rule you need to fix. Also, it is a good idea to setup a log monitoring system that can parse the large amount of logs easily.

### 5.3. Deny Mode

You can't stay in the log mode forever. When you are not seeing any more legitimate traffic getting logged by the SecRule during the log mode for a reasonable amount of time, you have to make a decision to go to the deny mode, which denies the requests that are matched by SecRule. To enable the deny mode, you need to change the "pass" variable to "deny" from the SetDefaultAction" line of the configuration file, as shown below. Again, if you have used any pass action for an individual rule which supersedes the default action, you need to change it to "deny, log, *redirect:127.0.0.1/dvwa/*" as well.

*SecDefaultAction "phase: 2, deny, log, redirect:127.0.0.1/dvwa/"*

When the request gets denied, it is wise to redirect the traffic to the default website instead of displaying error messages, because it is not appropriate to show to the customers an error message or to provide any clue to the attackers. As shown above, the "redirect" action is used to redirect the traffic to the homepage of the DVWA web application when the request gets denied. Even though you have spent reasonable amount time in the log mode, there is always a chance of misconfiguration or a new type of traffic to be denied. For example, there can be a critical process that only runs once a year, but it happens to include a string that gets denied by the existing SecRule. Therefore, it is important to prepare a system that monitors the denied requests and alerts the administrator immediately so he or she can check the logs to investigate the case.

## 6. Conclusions

Web applications have been evolving so fast and they have become one of the most important things that we can't live without, such as electricity and water. The use of web applications will not stop increasing but, on the other hand, attackers will not stop trying to penetrate your applications to. Implementing a web application firewall is a great method to protect your application from web attacks. However, the cost and the complexity of

implementing a WAF are huge. If you are new to the WAF technology, you should start with an open source technology, such as Modsecurity, to learn the technology. Then, as a next step, you can test your small application. Once you are confident with the technology, you can start implementing it for your main application in order to protect it.



Issac Museong Kim, iamissac@gmail.com

## 7. References

Damele, B., & Stampar, M. (2011). *Sqlmaps's user manual*. Retrieved from http://sqlmap.sourceforge.net/doc/README.pdf

Hansen, R. (2008). Xss cheat sheet. Retrieved from http://ha.ckers.org/xss.html

HP DVlabs, (2010). *2010 full year top cyber security risks report*. Retrieved from http://dvlabs.tippingpoint.com/img/FullYear2010Risk Report.pdf

Ivey, T. (2010). *Damn vulnerable web application official documentation*. Retrieved from https://dvwa.svn.sourceforge.net/svnroot/dvwa/docs/DVWA_v1.3.pdf

Mischel, M(2009). *Modsecurity 2.5*. (1st ed.). Birmingham, UK: Packt Publishing Ltd.

Modsecurity, (2011). *Modsecurity reference manual*. Trustwave Holdings, Inc.Retrieved from http://sourceforge.net/apps/mediawiki/mod-security/index.php?title=Reference_Manual

Owasp, (2011). *Web application firewall*. Retrieved from https://www.owasp.org/index.php/Web_Application_Firewall

Phongthiproek, P. (2011). *Beyond sqli: Obfuscate and bypass*. Retrieved from http://www.exploit-db.com/papers/17934/

Stuttard, D, & Pinto, M(2007). *The web application hacker's handbook*. (1 ed.). Indianapolis, IN: Wiley& Sons Publishing.

Security Compass. (2010). Retrieved from https://addons.mozilla.org/en-US/firefox/addon/xss-me/

Trustwave, (2011). *New modsecurity release includes key data protection advancements*. Retrieved from https://www.trustwave.com/pressReleases.php?n=new-modsecurity-release-includes-key-data-protection-advancements

Vela, E., & Lindsay, D. (2009). *Our favorite xss filters/ids*. Retrieved from http://www.blackhat.com/presentations/bh-usa-09/VELANAVA/BHUSA09-VelaNava-FavoriteXSS-SLIDES.pdf