Web Application Firewalls
Evaluation and Analysis

Andreas Karakannas
Andreas.Karakanas@os3.nl

George Thessalonikefs
George.Thessalonikefs@os3.nl

University of Amsterdam
System & Network Engineering MSc

June 1, 2014

Abstract

In order to overcome the security vulnerabilities of web applications most organizations are turning to Web Application Firewall (WAF) solutions. WAFs are residing between a web application and the network and they operate at the application layer (HTTP,HTTPS), providing better content filtering capabilities compared to the traditional firewall solutions. This report tries to document the effectiveness of Web Application Firewalls against the most well-known web application vulnerabilities, and the techniques used by WAF solutions for detection and prevention of those attacks.

Contents

1 Introduction

As the world of Internet is continuously evolving, the number of hosted web applications is continuously increasing. Bad programming practices during the application's development or bugs found in software applications (e.g., web servers), can pose the operation of the web application, the information stored in its database or the operation of the hosting machine at risk. In order to overcome the security vulnerabilities of web applications most organizations are turning to web application firewall (WAF) solutions. WAFs are residing between a web application and the network and they operate at the application layer (HTTP,HTTPS), providing better content filtering capabilities compared to the traditional firewall solutions.
Today, many open source WAF solutions are used based on out-of-the-box strength policies by IT-security engineers. In this project, we will focus on the effectiveness of open source WAFs against well-known web application attacks and we will analyze/correlate it with the methods/techniques that each WAF uses for finding and preventing these attacks.

1.1 Research Questions

Based on the aforementioned, we define the following main and sub research questions:

Main question:
How can the effectiveness of WAF solutions be correlated with the detection and prevention techniques?

Sub Questions:
What is the effectiveness of Web Application Firewalls against the most well-known web application vulnerabilities?

What techniques are used by WAF solutions for detection and prevention of these vulnerabilities?

1.2 Related work

As far as we investigated in related work, there is no scientific paper evaluating the effectiveness of the methods/techniques that are used by WAF solutions for detecting and preventing web applications attacks. Most of the related work [1], [2], [3] is done on evaluating WAF solutions in terms of security effectiveness, performance, stability, reliability, management and configuration. Web Application Security Consortium in the paper Web Application Firewall Evaluation Criteria [4] has defined a list of categories that has to be taken into account when evaluating WAF solutions. Among these categories the detection and prevention techniques importance is discussed and the most widely used techniques are listed.

2 Open Source WAFs

Through our research we came across various open source WAFs. Some of them were outdated (Guardian@JUMPERZ.NET, OpenWAF), others have gone commercial (Binary Sec), others are targeted to specific applications (WebCastellum, only for Java EE containers) or new solutions are still in the development phase (IronBee).

This list leaves us with the community's most sought after open source WAFs, ModSecurity for the Apache server and AQTronix's WebKnight for Microsoft's IIS.

Both WAFs depend on rules for their operation but the way each one uses and interacts with their ruleset varies greatly. In the following sections we will give a brief overview of how each WAF operates.

2.1 ModSecurity

ModSecurity is a module available for Apache that comes with no rules by default. This is because it offers a powerful rule engine that allows for extensive definition and customization of rules. But nonetheless OWASP, discussed in the next chapter, offers the OWASP ModSecurity Core Rule Set (CRS) which aims to "*provide an easily "pluggable" set of generic attack detection rules that provide a base level of protection for any web application*"[1]. From now on we consider an installation of ModSecurity including the OWASP CRS as a default installation.
The rules are written following the rule engine's specified format and can check request/reply headers/body and also customized variables against fixed values or regular expressions. The use of regular expression is extensive throughout the ruleset and the majority of the rules contain optimized regular expressions in order to speed up the processing time.

ModSecurity has two main modes of operation: Self-Contained and Collaborative Detection.

Self-Contained refers to the fact that if any rule will match against a request to the server, ModSecurity will block the request. This mode of operation is very strict and demands a fine-grained ruleset, customized specifically for a given application where false positives cannot be tolerated. It is not fitted for an out of the box generic default configuration.

Collaborative Detection on the other hand operates in a "delayed blocking" fashion. Each rule contributes to anomaly scores, inbound and outbound (depending on the malicious requests and information-leaking replies). Based on the significance of the rules they are divided into four categories with different points: critical, error, warning and notice. For a specific request or reply, rules that match increment the anomaly score until the threshold is reached and the request or reply is blocked.

Throughout this research we will use the second mode of operation as it is more suited for a generic default configuration regardless of the web application.

[1] https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project

2.2 AQTronix WebKnight

WebKnight is installed as a global filter on the server and comes with a default configuration present. It is in the form of XML files that can be visually represented with the WAF's own `Config` program. Through the program the user can easily interact with the configuration by altering the state of checkboxes, the default values of the parameters and the content of various lists that are used for matching against certain attacks (e.g., SQL keywords to match against an SQL Injection attempt).

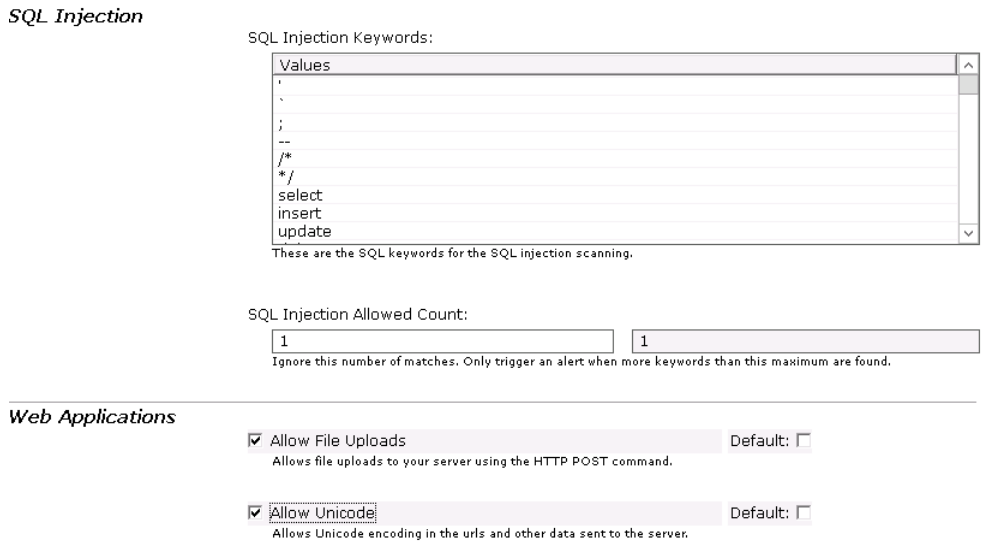

Figure 2.1: WebKnight's Config program

Although WebKnight does not offer a rule engine in order for the users to be able to craft their own rules, it is user friendly and allows for easy customization due to the human readable format of the configuration. Regular expressions are available in selected fields of the configuration and most of the sections contain fixed values that are part of a list.

WebKnight, with the extensive amount of options already available, offers the ability to customize its behavior for specific web applications, but if someone would have wanted to define new configuration options, he would have to do so by modifying the WAF's source code.

WebKnight's mode of operation is to block traffic that matches options that generate `ALERT` messages. `WARNINGS` are given in the form of log messages when less significant options have matched or when an option provides a threshold. In that case, for example the allowable total number of unique occurrences of SQL keywords in a request's parameters, until the threshold is reached only `WARNING` log messages are spawned.

3 OWASP Top Ten - 2013

The Open Web Application Security Project (OWASP) is a free and open community which main focus is to provide methodologies, documentation, tools and technologies on improving the security of web applications. As in every free and open community, everyone from individuals to corporations and educational organizations can participate in the running development and documentation projects thus contributing to the establishment of comprehensive best practices for application security.

One of the main projects of OWASP is the Web Application Firewall Evaluation Criteria (WAFEC)[2] in which a framework for effectively evaluating the strengths and weaknesses of WAF solutions is defined and documented. Among others, two of the most important selection criteria defined is the strength of default (out-of-the-box) defenses and the protection they offer against the OWASP Top Ten which we briefly describe in the next paragraph.

OWASP Top Ten is a project on which a variety of security specialists participating in the project share their knowledge and expertise in order to document a list with the top 10 most critical web application security risks. The top 10 risks are selected and prioritized based on vulnerability data gathered from hundreds of organizations and applications in combination with their exploitability, detectability and impact. This list is proposed by the OWASP organization as a guideline for the most important security vulnerabilities from which a web application needs to be protected from. As of today the most recent version of this list is the OWASP Top 10 - 2013. Below we list and briefly explain the top 10 security risks as defined in the OWASP Top 10 - 2013.

A1 – Injection: Injecting flaws, such as SQL, OS and LDAP injection is the most critical vulnerability. Injection occurs when untrusted commands/data are send through an HTTP request to an interpreter which in turn executes or stores this malicious data. This can have a very severe impact which spans from data loss and corruption to complete host takeover.

A2 – Broken Authentication & Session Management: Broken authentication and session management exploits take advantage of weaknesses in the authentication and session management functions in order to retrieve user and session critical information. This kind of vulnerability is severe for the users of a website.

A3 – Cross-Site Scripting (XSS): XSS occurs when untrusted functions are sent and executed to a user's web-browser, leading to session hijacking, redirection to malicious sites, deface web sites and more. There are two types of XSS attacks, stored XSS which is persistent (e.g., present in the web site's database and can be executed for every user that views the XSS content) and reflected XSS, which is included in a URL which can be sent to the victim through phishing emails for example.

A4 – Insecure Direct Object References: Insecure direct object references occur when a website exposes object references to the user. An attacker can take advantage of this vulnerability by changing the object reference value and gain access to other user accounts for example.

A5 – Security Misconfiguration: Default security settings and unupdated software of the application server, web server and database server which leaves backdoors for the attacker.

[2] https://www.owasp.org/index.php/Web_Application_Firewall

A6 – Sensitive Data Exposure: Sensitive data exposure occurs when security critical data such as credit cards numbers are not given special protection such as encryption when exchanged with the user's web-browser.

A7 – Missing Function Level Access Control: Refers to the fact that a web application may require user authentication but fails to detect the level of access a user can have for parts of the application. Authorization is not handled properly and a result could be that a normal user has access to administrative only functions.

A8 – Cross-Site Request Forgery (CSRF): CSRF is closely related to XSS and occurs when XSS requires for a user to be already authenticated in a given web application. XSS could then be used in order to send an HTTP request that the browser will automatically accompany with any relevant cookies (e.g., session ID) it may have. If for example this request is a request for changing a user's password then the user will be compromised and the attacker will now take-over that user's account.

A9 – Using Components with Known Vulnerabilities: Web applications that use libraries, frameworks and other software modules that have known vulnerabilities are automatically vulnerable to these vulnerabilities.

A10 – Unvalidated Redirects and Forwards: Unvalidate redirects and forwards occur when a website redirects the users to another destination page without proper validation. Attackers can take advance of this unvalidated redirect and direct the users to phishing or malware site.

Taking into account the aforementioned open source WAF solutions and the OWASP Top 10 – 2013, we focused on evaluating and analyzing the effectiveness of ModSecurity and WebKnight's out-of-the-box defenses against well-known attacks on web application security risks defined in the OWASP Top 10 -2013.

4 Approach and Scope

4.1 Experimental Environment

For the purpose of our project we set up the environment shown on Figure 4.1. On one hand we set up an Ubuntu 14.04(Trusty) server and on the other hand we set up a Microsoft Windows 2012 Server. On the Ubuntu server we set up Apache 2.4.7 and on the Windows server we set up Internet Information Services(IIS) 8.5 as the web servers. On both servers MySQL Community Edition version 5.6 and PHP version 5.5 were installed and configured. ModSecurity and WebKnight were installed and configured on Apache and IIS respectively. As already mentioned ModSecurity makes use of OWASP's CRS and WebKnight uses the default configuration.

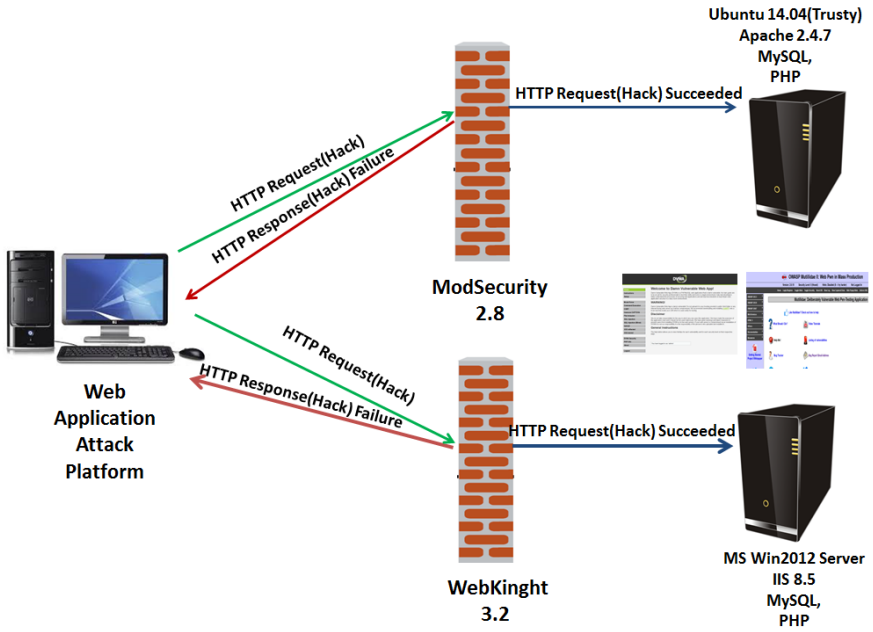

Figure 4.1: Experimental environment

Damn Vulnerable Web Application (DVWA) and Mutillidae were used as the vulnerable web applications that the WAFs were protecting. Both applications are suitable for performing a number of web attacks and are widely used for ethical hacking educational purposes. A desktop computer was used as the application platform from which all the attacks were performed.

The experimental environment described above was created in a controlled and isolated environment that was not accessible by third parties. Specifically, for the deployment of the webservers we used two virtual machines running on a server provided to us from OS3 and accessible only from the desktop machines from which the attacks were performed.

4.2 Approach

In order to evaluate the effectiveness of the WAFs under investigation and analyze their prevention methods against the attacks for exploiting OWASP TOP 10 – 2013 security risks we followed the steps explained below.

Initially, we attempted a number of attacks without the presence of a WAF and we documented the results. The attacks that managed to exploit the vulnerabilities of the web applications were used later against the WAF solutions that we investigate. We try to implement at least one successful attack for each OWASP TOP 10 – 2013 category. Due to the fact that the web applications that we used did not include all of the OWASP Top 10 – 2013 security risks, the difficulty in exploiting some of the vulnerabilities and the limited amount of time during this project we did not manage to cover all the security risks defined in OWASP Top 10 – 2013. The security risks for which we managed to find at least one attack that exploited them are shown in the table below.

OWASP Top 10 - 2013	Successful attack performed
A1 – Injection	√
A2 – Broken Authentication and Session Management	√
A3 – Cross-Site Scripting	√
A4 – Insecure Direct Object References	√
A5 – Security Misconfiguration	√
A6 – Sensitive Data Exposure	?
A7 – Missing Function Level Access Control	x
A8 – Cross-Site Request Forgery	√
A9 – Using Components with Known Vulnerabilities	x
A10 – Unvalidated Redirects and Forwards	x

Table 4.1: OWASP Top 10 – 2013, successful attacks performed

Note: *There was no data available to test for A6 – Sensitive Data Exposure. Nonetheless the default configurations of the WAFs related to this vulnerability are discussed.*

After the list of the successful attacks was created, the next step was to perform these attacks against ModSecurity and WebKnight. For each attack we observed the effectiveness of detection/prevention of each WAF. Further analysis was performed in order to find out what methods/mechanisms are used by each WAF against each type of attack. The list of the attacks that were conducted against the WAFs along with any tool that may be used are aggregated in Table 4.2 and explained in detail in the following sections.

OWASP Top 10 - 2013	Attack performed	Tool used
A1 – Injection	SQL injection	Sqlmap
	Command injection	Manually
A2 – Broken Authentication and Session Management	Bypass authentication with SQL injection	Manually
	Bypass authentication with brute force	Hydra
A3 – Cross-Site Scripting	XSS stored	Manually
A4 – Insecure Direct Object References	Modify object reference value	Cookie Manager
A5 – Security Misconfiguration	Upload and execute php file	Manually
A8 – Cross-Site Request Forgery	CSRF to change user credentials	Manually

Table 4.2: Web attacks performed and tools used

SQL Injection (SQLi): Sqlmap was used in order to automate the SQLi attacks against vulnerable pages of the web applications. Sqlmap also offers various `tamper` scripts that will change the encoding of the request in order to bypass detection by security defenses.

Command Injection: Command injection was performed manually in pages where the applications allowed communication with the underlying operating system (e.g., pinging an IP address). Our goal was to gather information about the underlying operating system and attempt to execute programs on the host that could give us unauthorized access(e.g., opening a listening port using a preinstalled `nc`).

Bypass authentication with SQL Injection: We tried manual SQLi in the password field of a login form in order to bypass authentication. Different SQLi parameters and encodings were used in order to try to bypass the WAF.

Bypass authentication with Brute Force: `THC-Hydra` was used to perform brute forcing of a login form using a custom dictionary. The dictionary had 500 passwords in total with the valid password being the last entry.

XSS Stored: We attempted to store an XSS in the database by passing JavaScript code in text fields that will later be viewed (thus, ran) by other users of the site.

Insecure Direct Object References: In this attack we modified the value of an insecure object reference that was returned as a cookie to a logged in user. Manipulating the value of this object reference using `Cookie Manager` gave access to other user accounts (e.g., log in as admin).

Upload and execute php file: Using `metasploit` we created a `php/meterpreter/reverse_tcp` payload. We uploaded and executed it on the server (the directory of the uploaded files was exposed to the users) in order to create a reverse tcp connection and gain shell access.

Cross Site Request Forgery: For this type of attack we used an authenticated cookie along with a URL GET request in order to imitate a CSRF attack where the victim would click on a given URL found in phishing emails for example. The result of the GET request used was to change the password on the logged in site.

Evaluating and analyzing the results of the above attacks gave us insights of the effectiveness and the methods/mechanisms each WAF uses for detecting/preventing the aforementioned attacks.

5 Results & Evaluation

In this chapter, we present and analyze the results of our attacks described in the previous chapter. For each attack, we document the effectiveness and the mechanism that each WAF solution uses for detecting and preventing the attack. Table 5.1 shows an overview of our results. A detailed description of each result then follows.

OWASP Top 10 – 2013	Attack	Results	
		ModSecurity	WebKnight
A1 – Injection	SQL injection	Blocked	Bypassed
	Command injection	Blocked	Blocked
A2 – Broken Authentication and Session Management	Bypass authentication with SQL injection	Blocked	Bypassed
	Bypass authentication with brute force	Bypassed	Bypassed
A3 – Cross-Site Scripting	XSS stored	Blocked	Bypassed
A4 – Insecure Direct Object References	Modify object reference value	Bypassed	Bypassed
A5 – Security Misconfiguration	Upload and execute php backdoor file	Bypassed	Blocked
A8 – Cross-Site Request Forgery	CSRF to change user credentials	Blocked	Bypassed

Table 5.1: Results overview

Before continuing with the results it should be noted that both WAFs were reporting warnings that eventually led to blocking our traffic. This was due to the Host HTTP header not containing a Fully Qualified Domain Name (FQDN), or the User-Agent HTTP header containing the name of the security tool used. Both checks were bypassed on purpose by inserting an entry into `/etc/hosts` and using the tool's options or a proxy in order to change the User-Agent HTTP header in something commonly used by a mobile device for example, respectively.

5.1 A1 – Injection

SQL Injection

ModSecurity: All the attempts of SQL injection, also with the use of the various tamper scripts, were unsuccessful. ModSecurity offers an extensive list of regular expressions that filter request parameters for possible SQLi attacks.

WebKnight: When running the same test against WebKnight we had one successful attempt while using the '`apostrophenullencode`' `tamper` script. The aim of the script is to substitute the apostrophe (') with its illegal double Unicode counterpart (`%00%27`). This value is not present in WebKnight's blacklist for SQLi and because of the default configuration that allows one item of the SQLi list to be uniquely present in the request, sqlmap successfully gained access to the database. An example of a successfully bypassed SQLi attempt follows:

Normal SQLi: `' or '1'='1`

Apostrophenullencode SQLi: `%00%27 or %00%271%00%27=%00%271`

In this attempt WebKnight detects the use of '`or`' but the double Unicode encoding for the apostrophe goes undetected.

Command Injection

ModSecurity: Command injection was possible but limited. ModSecurity has a blacklist of keywords used in general attacks containing among others names of well known executables (e.g., `uname`, `shutdown`, `telnet`) and well known files and directories (e.g., `etc`, `passwd`, `httpd.conf`). Added limitation comes into play when one wants to include path traversal in order to reach a file, because of the WAF's detection of '`../`'. However with selected commands such as `ifconfig` and `ip route` we were able to gather information about the web server.

WebKnight: The same mechanism, using a blacklist, is applied to WebKnight as well. However, while the list contains and blocks known executables (e.g., `cmd.exe`), it does not include '`cmd`' for example which has the same meaning as '`cmd.exe`' in Windows environments. Moreover, path traversal in a list of allowed paths (e.g c:\inetpub\www) is allowed, thus an attacker may be able to access critical files if by mistake the administrator has stored them in a directory inside the allowed paths scope.

5.2 A2 – Broken Authentication and Session Management

Bypass authentication with SQLi

ModSecurity: We were not able to bypass authentication as ModSecurity is quite resistant to SQLi attempts as shown earlier.

WebKnight: We were able to bypass authentication with a simple substitution of 'or' with '||' in our SQLi query. We noticed that logical symbols such as '`||`' and '`&&`' were absent from the SQLi keywords. Again, we were successful because of the default configuration that allows one item of the SQLi blacklist to be uniquely present in the request, in our case only the apostrophe was detected.

Bypass authentication with Brute Force

ModSecurity: By default ModSecurity offers no protection against brute force attacks. Nonetheless, there is an experimentation rule present that needs to be activated when such protection is desired. It can monitor requests for specific (login)pages and block IP addresses for a customized period of time if a customized ratio of number of requests/time is exceeded. However, this rule is not an optimal solution as it could block an IP address corresponding to a NAT gateway and eventually emulate a DOS attack on the server for the users of that private IP space.

WebKnight: A similar blocking mechanism as the one used by ModSecurity is available by default on WebKnight but it is only effective on pages using HTTP Authentication methods which return a '`401 – Unauthorized`' status code when authorization fails. However, most web applications use `GET` and `POST` requests in order to pass login credentials and check them against stored values in the database instead of relying on the web server's configuration.

5.3 A3 – Cross-Site Scripting (stored)

ModSecurity: ModSecurity managed to block all the attempts to insert stored XSS into the database. Due to the fact that ModSecurity has an extensive list of regular expressions that filter the parameters in the request header for almost all possible types of encoding, there is no possible way to bypass it.

WebKnight: WebKnight also managed to block most of the attempts for inserting the XSS into the database. Specifically, inserting an XSS using the default encoding for scripting (e.g., `<script>alert(“XSS stored”)</script>`) is blocked. WebKnight uses an extensive list of keywords (e.g., `<script`, `<applet`, `<object`) that when one of those is matched in the content of the HTTP request, the request is immediately blocked. Moreover, it can detect these keywords even if the attacker uses different encoding (e.g., `Base64`, `Hex`, `Octal`, `Binary`, `ASCII hex`).
However, MySQL encoding is missing from the encoding schemes that WebKnight uses and therefore an XSS can bypass WebKnight if it is passed to the HTTP request in MySQL encoding (e.g., `char(60,115,99,…)`).

5.4 A4 – Insecure Direct Object References

ModSecurity: ModSecurity was unable to detect manipulation in the value of the insecure object reference that was returned to the user as a cookie. ModSecurity has no mechanism to prevent this kind of attack as it is heavily dependent on the web application’s logic.
However, given the ability to create custom rules this attack could be prevented. In order to prevent this kind of attack, every live session’s critical cookies that lead to insecure direct object references can be stored by ModSecurity. Every time a user makes a new HTTP request and pass his cookies values associated with his current session ModSecurity can check the values of the cookies with the values stored on the server, thus manipulation of an insecure object reference value can be detected. A similar technique is used to prevent CSRF as discussed further in the report.

WebKnight: WebKnight also was unable to detect the manipulation in the insecure objects’ reference value. WebKnight has also no mechanism for detecting this kind of attacks and due to the limitation it offers in creating new rules the only way to implement a mechanism for this vulnerability is to add it to the source code.

5.5 A5 – Security Misconfiguration

Upload and execute php backdoor file

ModSecurity: ModSecurity failed to prevent uploading the php executable and also failed to block the execution of the backdoor file. ModSecurity has no mechanism by default for preventing this kind of attacks but as mentioned before a rule that could detect and prevent this attack can be used along with Apache’s ‘`Location`’ tags to scope the rule in certain upload forms and directories where these uploads could be found.

WebKnight: WebKnight managed to block uploading a possible malicious file to the host server. It filters the contents of the file that it is uploaded and if it matches a list of keywords (e.g., `<?php`) that are considered harmful it immediately blocks the request.

5.6 A6 – Sensitive Data Exposure

ModSecurity: Regarding sensitive data exposure ModSecurity offers regular expressions to match GSA, MasterCard, Visa, American Express, Diners Club, Discover and JCB credit card numbers in the server's replies.

WebKnight: WebKnight also offers a list of regular expressions to match Visa, American Express, Mastercard and Discover in the server replies. This list can be extended by the administrator to include more sensitive data that need to be protected.

5.7 A8 – Cross-Site Request Forgery

ModSecurity: By default no countermeasure is enabled but there is an optional rule for CSRF protection available making use of the CSRF Token. CSRF Token is a mechanism used by ModSecurity in order to prevent CSRF attacks. It works by creating and storing (on the server) a CSRF Token whenever a new session is initiated. It then uses Content Injection in order to inject JavaScript code on the replies generated by the server. This JavaScript code will update HTML forms in order to include the CSRF Token when submitted. It will also update any HTTP links found in the '`src`' and '`href`' attributes of HTML tags to include the CSRF Token in the GET parameters.
This way a request coming from a cookie-authenticated user will also need the CSRF Token's value in order to be valid. By using this mechanism an attacker can no longer target the whole site's userspace with a single phishing email for example, as each user session has its own CSRF Token. He needs information on the CSRF Token which is stored on the server. The CSRF Token could also be sniffed if the connection to the site is unencrypted. Even then he could only attempt a CSRF attack on a specific victim within a specific session.
As noted by the authors, this implementation does not work currently with AJAX and should not be used globally. One suggestion is to use it on post-authentication directories using Apache's '`Location`' tags.
In our tests we found out that the CSRF Token was applied correctly when using the Chromium web browser, but we were unable to make it work with Mozilla's Firefox. Further research into this was not conducted due to time limitations, but we can also derive that the implementation is browser dependent.

WebKnight: No detection or blocking was reported by WebKnight. WebKnight does not support CSRF Token which is the de facto method for preventing CSRF attacks. It uses a deny Referer list in which the administrator can add the domains (FQDN) or IP addresses that are not allowed to be at the HTTP Referer field of the HTTP request. However, this is not a full proof countermeasure as the Referer HTTP header can easily be spoofed.

6 Conclusion

In this research project we looked at open source Web Application Firewalls and the effectiveness of their detection and prevention mechanisms against well known web vulnerabilities. The WAFs under evaluation were ModSecurity and AQTronix's WebKnight. Both of these WAFs were used out-of-the-box based on their default configuration. In the case of ModSecurity that comes with no rules the standard OWASP Core Rule Set was used.

Both WAFs were tested against vulnerabilities present in OWASP's Top 10 – 2013 list. The majority of the attacks against these vulnerabilities were blocked by both WAFs. In cases where an attack was successful the means to block it were available in the WAF but needed extra configuration.

We noticed that in comparison, ModSecurity is more dynamic and customizable but also harder to manage through the various configuration/rule files. WebKnight on the other hand does not offer the level of customization that ModSecurity offers but is more easily manageable through the user friendly '`Config`' accompanying program.

If one requires further support, consultancy can be provided by AQTronix or third parties for WebKnight and ModSecurity respectively at a fee.

For every WAF deployment a period of passive auditing in order to detect and resolve false-positives and false-negatives is recommended before using the WAF in active security mode.

WAFs in general offer a convenient solution for protecting legacy or black-box web applications against evolving threats. But as we have already encountered vulnerabilities except from technical vulnerabilities, present themselves often in the web application's logic layer. In that layer a WAF is not able to by default detect and block such attacks.

Based on these observations we conclude that a WAF should be used as a last step security solution and should not be regarded as the main security net for the underlying web application. Security concerns should be addressed as best as possible during the development phase of a web application.

References

[1] SACHIN. Top 10 open source web application firewalls (waf) for webapp security. 2011. URL: http://www.fromdev.com/2011/07/opensource-web-application-firewall-waf.html.

[2] MILLER, Sandra. Reviews of six top web application firewalls. *Information Security Magazine*, 2008. URL: http://searchsecurity.techtarget.com/ezine/Information-Security-magazine/Reviews-of-six-top-Web-application-firewalls.

[3] SUTO, Larry. Analyzing the effectiveness of web application firewalls.*IM-PERVA, Redwood Shores CA*, 2011. URL: http://www.ntobjectives.com/files/Analyzing-the-Effectiveness-of-Web-Application-Firewalls.pdf.

[4] AUGER, Robert, et al. Web application firewall evaluation criteria.*Web Application Security Consortium*, 2009. URL: https://baoz.net/web-application-firewall-evaluation-criteria/.