Profiling Web Applications for Improved Intrusion Detection

GIAC (GCIA) Gold Certification

Author: Manuel Leos Rivas, MLeosRivas@mastersprogram.sans.edu

Advisor: Richard Carbone

Accepted: September 5, 2016

Abstract

Web application firewalls using generic "out of the box" configurations work well for common vulnerabilities but lack the capability to address application-specific contexts. Due to this lack of context, it is difficult for the firewall to determine what it is 'good' versus 'bad'. In addition, several learning features of certain high-end devices are inaccessible to companies and individuals. This document provides a generic approach to protecting web applications using freely available software by configuring ModSecurity. This approach enables differentiation between what is acceptable for the application and what may be interesting for investigation purposes. The process for creating an application profile should be well documented, repeatable, verifiable and automated as much as possible to ease integration into the application development lifecycle.

1. Introduction

Most web application firewalls (WAFs) inspect requests and responses by searching for known abnormal or suspicious patterns. Achieving a good balance between false positives and false negatives depends on the type and content of the web application. Such a balance is difficult to achieve because applications encompass a wide variety of content and encoding. WAFs that lean toward ease of implementation often use a much more relaxed inspection policy, which may give a false sense of security. Thus, a WAF's effectiveness is greatly affected by its rule set and the context of the underlying application.

Defining a profile describing the common usage of a web application is crucial for identifying deviations or anomalies. Enhancing the intrusion detection process using both positive and negative security approaches makes WAFs resilient to attack variations and helps reduce false positives.

A detailed application profile can identify the deviations from normal usage for an application. Such variations can be used to detect changes in the application, different data or types of data, data locations, etc. Profile creation linked to the data acquired during testing or monitoring is an iterative process that should be performed with each new version of the application. Thus, automation is a desired feature due to the number of times the actions will need to be performed.

Generating a profile that can later be transformed into WAF rules can be automated, but it is highly advised to supervise the output to prevent the WAF from generating false negatives. The extent to which the process and tools are automated will allow it to scale up from small deployments to enterprise systems.

Many commercial tools and vendors provide features similar to those found in open source tools. The open source tools examined in this paper include ModSecurity (SpiderLabs, ModSecurity Open Source Web Application Firewall, 2016) and OWASP CRS (OWASP, OWASP ModSecurity Core Rule Set Project, 2016) for WAF capabilities, OWASP ZAP (OWASP, OWASP Zed Attack Proxy, 2016) as an application proxy, zap2modsec (Barnett R., 2013) as a virtual patch generator and

WebAppProfiler (Leos Rivas, 2016), simplerules, profileeditor and other Python scripts to automate profile and rule creation.

2. Web application profiling

Profiling a web application entails describing all the elements and types of exchanges of the application. The idea is to understand what is normal as much as possible so that alerts can be built into the WAF to detect anomalies.

Every change in the application requires a profile update to reduce false positives. Thus, automation plays a major role in the process because complex web applications may contain thousands of elements.

"The Web Application Profile constitutes a high-level XML description of web applications which serve as a basis for positive security models for the applications." (Bockermann, 2007)

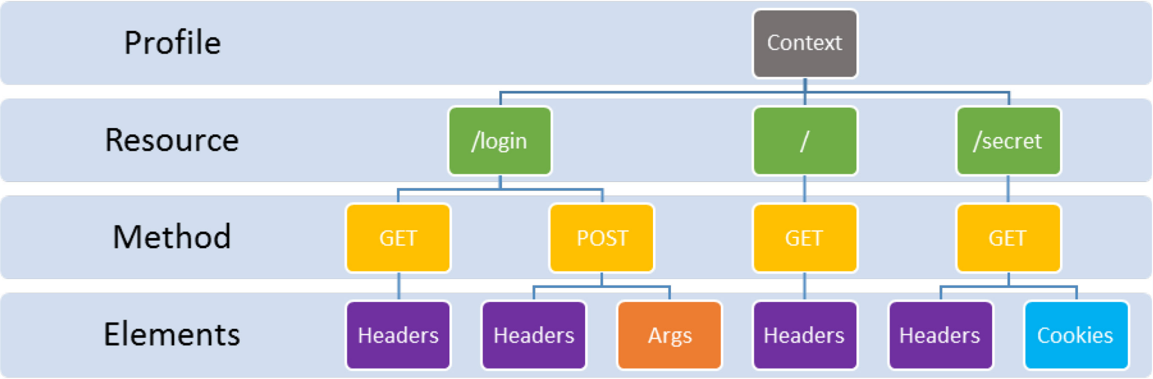
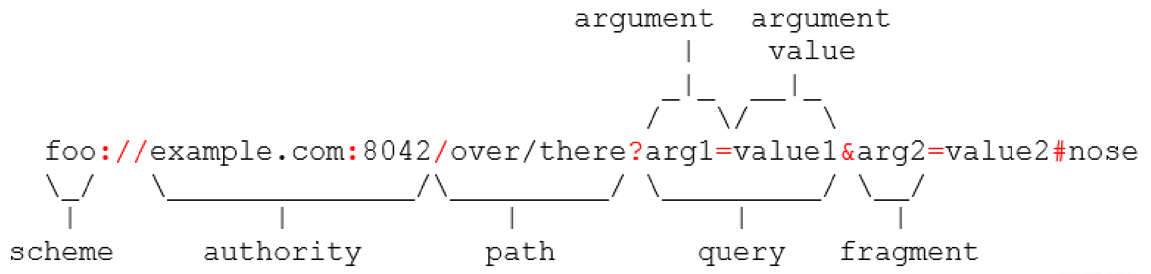


Figure 1 : Graphical representation of a basic profile

2.1. Elements used for profile creation

ModSecurity parses requests and responses to create variables to address every item in an application. Each application has a particular set of characteristics that can be used to create this context, including the uniform resource identifier (URI), method, arguments, HTTP headers, cookies and body.



```
                                 argument  argument
                                      |      value
                                     _|_    __|_
                                    /   \/      \
  foo://example.com:8042/over/there?arg1=value1&arg2=value2#nose
  \_/   \______________/\_________/ \___________/ \__/
   |            |            |            |          |
scheme      authority       path        query    fragment
```

Figure 2 : URI structure, see IETF RFC 3986 (Berners-Lee et al., 2005) and RFC 7320 (Nottingham, 2014).

2.1.1. Uniform Resource Identifier (URI)

The URI pattern depends on the application behind the WAF; it may be static with a unique URI or dynamic with a single URI with called arguments, or it may build the resource on demand. It is important to identify which resource is being accessed, where it is being accessed from and by whom. Many applications perform sensitive operations that should not be made available on the Internet or to the general public. Application profiles typically need to include all the different URIs listed in the relevant logs or database of connections as a basis for building listings of characteristics. Private characteristics must be flagged to provide special alerts. For dynamic sites, a list that includes regular expressions may be a better approach.

Some web or application servers have their own “hidden” functions such as administrative consoles or status pages. The lack of a direct link to a URI on a website does not necessarily mean that it does not exist. FuzzDB (Muntner , 2010) lists several common types of resources and hidden pages.

The list of URIs can be used to deny access to private resources. Multiple lists can be included such as common or public, private or sensitive, banned or fake resources. Not all events are identical -- there should be no alarms on public resources, and events on banned resources should respond automatically by closing the user session, banning the user or source for a given period of time, or any other deceptive action or countermeasure.

Each URI is described as a *Resource* element in the XML profile.

2.1.2. HTTP request method

The GET and POST methods are used in virtually every web application. Other commonly used methods include HEAD, OPTIONS, PUT and DELETE. Some applications may use a combination of GET and POST, but they are not interchangeable.

The most common method for transferring information to the server consists of either using arguments in the request line as part of the URI (GET method) or as part of the body (POST method). The earlier method is riskier because the transferred information visible in the request line may by bookmarked, cached or logged; thus, it cannot be used for sending sensitive data (personal, financial, credentials, etc.).

The profile should list the methods allowed by the entire application as well as the specific methods used to access each URI.

Each HTTP request method is described as a *Method* element nested under the respective *Resource* element in the XML profile.

2.1.3. Arguments

ModSecurity can address the arguments by name. In a GET request, the arguments are present in the query string. A POST request is part of the body, so it will only be parsed by default if the content type is application/x-www-form-urlencoded with the “urlencoded” processor. This is not the case for all other content types unless they are explicitly set to use a specified processor such as JSON or XML.

The profile must list the arguments and expected types of values for each URI and method. Values that are mandatory, sensitive or intended to be sanitized from the logs should be stated as such in the profile.

A WAF can identify arguments as evil payloads depending on their contents as it applies generic rules, causing certain words or symbols to block the request. A well-designed validation pattern included in the profile outperforms a long list of banned elements. The validation patterns of complex fields such as passwords and free text fields may be challenging, and international characters further complicate matters, even if the profile is forced to use generic wildcards. Fortunately, a maximum input length can be set.

Each argument is described as a *Parameter* element nested under the respective *Method* element in the XML profile.

2.1.4. HTTP headers

HTTP headers serve several different functions, and in some applications they are used for security purposes.

Specific headers have a significant impact on how a transaction is handled by the server. For example, the Host header causes the server to serve particular content, while other headers will cause the WAF to handle the request differently, i.e., Content-Type defines how a request is to be parsed. Poorly configured WAFs may trust these headers and disable further inspection. In text-based content, problems arise when the application behind the WAF interprets it incorrectly, leading the filter to allow a bypass and cause potentially serious consequences. Some headers affect how a client handles a request such as via content security policy or X-XSS-Protection.

Special headers such as Proxy, X-Forwarded-For, X-Originating-IP, X-Remote-IP, and X-Remote-Addr may have undesired results in the destination application and the WAF. Headers consisting of any other type of user input cannot be trusted without verification. Response headers (OWASP secure headers) may help to improve security.

The profile must include the headers and content types required to be present in each URI request. If the header is sensitive, it must be sanitized from the logs and responses (X-Powered-By forbidden, ASP-NET forbidden, X-AspNet-Version, etc.). As there is a wide variety of clients with their own set of headers and values, non-required headers may be dropped if they are not required by the application.

Each HTTP header is described as a *Header* element nested under the respective *Method* element in the XML profile.

2.1.5. Cookies

The cookies that are exchanged between the client and server have important flags -- such as *httponly* and *secure* -- that tell the client that a cookie value has restricted access (e.g., no *JavaScript* access) and should only be sent over encrypted channels. In addition, the domain, path and validity properties should be restricted when these types of cookie values are sent to the server.

Session cookies are used to track a session over multiple requests, track the client's actions and grant access. Some WAFs offers a session tracking feature to validate a session's validity. ModSecurity uses the `setsid` action to identify a transaction as part of a session.

The profile should list the cookies present in each request per URI in addition to all the relevant flags. If a cookie path is set to "/", it will be present on every resource. Because this value is generic, there is no value in adding this attribute to the profile. However, cookies used to identify sessions must also contain a "`SessionCookie`" element so they can be used in the setup and tracking of the HTTP session.

Each cookie will be a *Cookie* element nested under the respective *Method* element in the XML profile. The session cookies have an additional entry as a *SessionCookie* element at the same level.

2.1.6. Body

Depending on the type of application, the request body can range from zero to several kilobytes, or higher. The content may include text, encoded, XML, JSON, base64 encoded or multiple encoded types. Many applications send more data to the client than what they receive, although other applications send less data to the client than what they receive.

To perform the necessary decoding or transformations for handling anomalies, the profile can list the request and response body characteristics per resource or URI, such as maximum size, content type and encoding.

The arguments in an HTTP POST method are in the HTTP request body and must be specified as "`Parameter`" elements with the attribute "`scope`" set to "`body`".

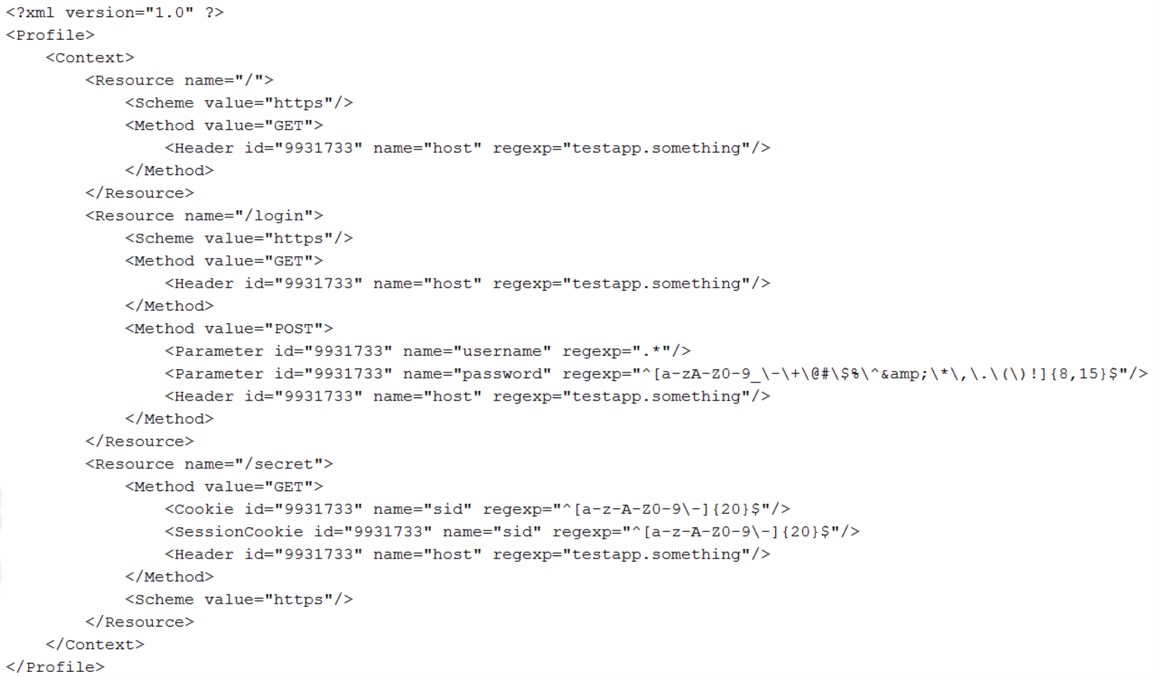
2.1.7. HTTP response code

Response codes vary depending on the application at hand. Most applications include 200 for normal responses, 404 for content not found, 403 for forbidden and 500 for server errors. Dynamic applications may use different or modified codes to send valid responses.

The profile must include the allowable response codes. A different response code may tell the WAF to redirect to a standard error page and prevent the leaking of valuable information such as code dumps or other intelligence leaks.

2.2. Creating the web application profile

The web application profile can be manually created using any XML editor, assisted with the profileeditor Python script or Web Profile Editor (Bockermann, Web Profile Editor, 2007) or automatically generated using WebAppProfiler, which extracts relevant elements to build profiles from ZAP sessions or ModSecurity audit logs. When an automated process is used, a manual review can be performed to adjust the elements to prevent false positives and false negatives.



```
<?xml version="1.0" ?>
<Profile>
    <Context>
        <Resource name="/">
            <Scheme value="https"/>
            <Method value="GET">
                <Header id="9931733" name="host" regexp="testapp.something"/>
            </Method>
        </Resource>
        <Resource name="/login">
            <Scheme value="https"/>
            <Method value="GET">
                <Header id="9931733" name="host" regexp="testapp.something"/>
            </Method>
            <Method value="POST">
                <Parameter id="9931733" name="username" regexp=".*"/>
                <Parameter id="9931733" name="password" regexp="^[a-zA-Z0-9_\-\+\@#\$%\^&amp;\*\,\.\(\)!]{8,15}$"/>
                <Header id="9931733" name="host" regexp="testapp.something"/>
            </Method>
        </Resource>
        <Resource name="/secret">
            <Method value="GET">
                <Cookie id="9931733" name="sid" regexp="^[a-z-A-Z0-9\-]{20}$"/>
                <SessionCookie id="9931733" name="sid" regexp="^[a-z-A-Z0-9\-]{20}$"/>
                <Header id="9931733" name="host" regexp="testapp.something"/>
            </Method>
            <Scheme value="https"/>
        </Resource>
    </Context>
</Profile>
```

Figure 3 : XML representation of a basic profile

As a python script, WebAppProfiler can be easily modified to change the XML schema to be compliant with either the Bockermann (Bockermann, 2007) or Ristic & Shezaf (Ristic & Shezaf, 2008) models. However, the initial version of the script produces an XML profile compatible with the earliest tools such as jwall web application profiler (Bockermann, 2007), and Web Profile Editor uses a graphical interface that can be used to produce profiles manually but is not conducive bulk or automatic processing. In addition, graphically designing the profile is much slower than scripting short commands. The WebAppProfiler and profileeditor scripts may be used as modules to further automate the process or used as APIs for graphical interfaces.

2.2.1. Automated data collection

For the initial application data collection for the present study, a testing environment was set up with the Apache httpd 2.2 or 2.4 server (The Apache Software Foundation, 2016, Apache HTTP Server Project), ModSecurity 2.9.0 or later, ZAP 2.5, python 2.7 (The Python Software Foundation, 2016, Python 2.7), the required python-owasp-zap-v2.4 API libraries (The Python Software Foundation, 2016, Python OWASP ZAP API), HTTP-parser (The Python Software Foundation, 2016, Python HTTP-parser) and tabulate (The Python Software Foundation, 2016, Python tabulate). The aim was to test all the functionalities and use cases of the application as opposed to testing the modsecurity rule set itself; thus, having a dedicated virtual host for this purpose was highly worthwhile. The term Virtual Host refers to the practice of running greater than one website (such as company1.example.com and company2.example.com) on a single machine. “Virtual hosts can be *IP-based*, meaning that you have a different IP address for every website, or *name-based*, meaning that you have multiple names running on each IP address. The fact that they are running on the same physical server is not apparent to the end user” (The Apache Software Foundation, 2016, Apache Virtual Host documentation).

Virtual hosts are required to have an independent audit log, and the ModSecurity engine must be enabled. Configuring the host to apply/withhold rules prevents false positives. In the testing environment, the ModSecurity objective is to summarize all the requests and response elements to create a positive security model. The rules created with the model cause warnings or blocks if anomalies are detected, and a second testing virtual host is recommended for deploying any rules produced after processing the profile to check that navigation is unaffected before deploying the rule set in the production environment.

A server or virtual host should be connected to the internet only after deploying basic access controls and best practice techniques following Apache security tips (The Apache Software Foundation, 2016, Apache security tips) and performing general security hardening of the server (CIS, 2016) or working with an isolated host-only virtual machine.

2.2.2. Performing application validation for profiling

Using an application proxy during the application validation phase can provide an additional log point wherein additional checks can be performed to passively search for defects. The resulting rule set can have virtual patches for any detected vulnerabilities. It is desirable to avoid polluting the test results so the process can be automated; however, rules that accept attacks or malformed requests should not be generated. If an exclusive virtual host is not available, a control cookie or header must be added to subsequently filter out the desired requests from the audit log or the zap persistent connection database.

For applications using Transport Layer Security (TLS), the application proxy handles the connection by impersonating the target site using a self-generated certificate, trusting the proxy certificate's ability to prevent connection issues. Some browsers and sites may detect that the certificate is not original. In such cases, the cache and history are cleared for the connection to succeed.

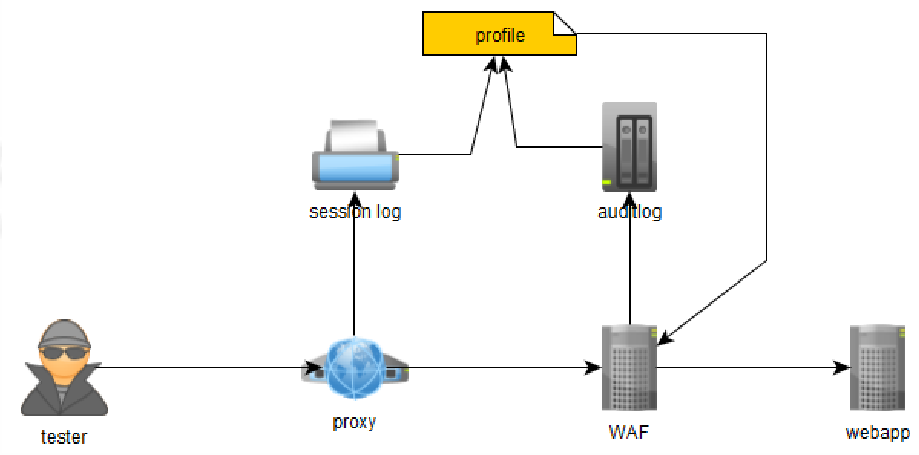
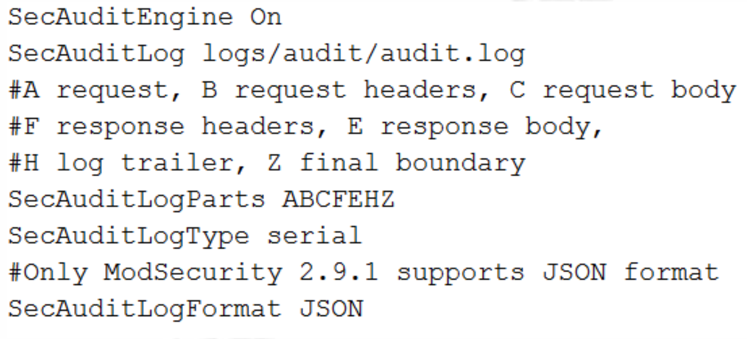


Figure 4 : Sample connection for the testing environment

2.2.3. Configuring the ModSecurity audit log for profiling

As an alternative to ZAP, a ModSecurity audit log can be configured to log every request or capture all the requests associated with regular application usage. The audit log is the most stable source available for profiling purposes and is perfect for lengthy or heavy testing.

The serial log format of ModSecurity is much easier to parse and script because every transaction is sent into the same file instead of the concurrent type format. The JSON format is supported in ModSecurity version 2.9.1 and beyond. “JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate” (Crockford., 1999). The WebAppProfiler script parses the audit log to rebuild messages to their original state by tracking the different log parts of every transaction id.



```
SecAuditEngine On
SecAuditLog logs/audit/audit.log
#A request, B request headers, C request body
#F response headers, E response body,
#H log trailer, Z final boundary
SecAuditLogParts ABCFEHZ
SecAuditLogType serial
#Only ModSecurity 2.9.1 supports JSON format
SecAuditLogFormat JSON
```

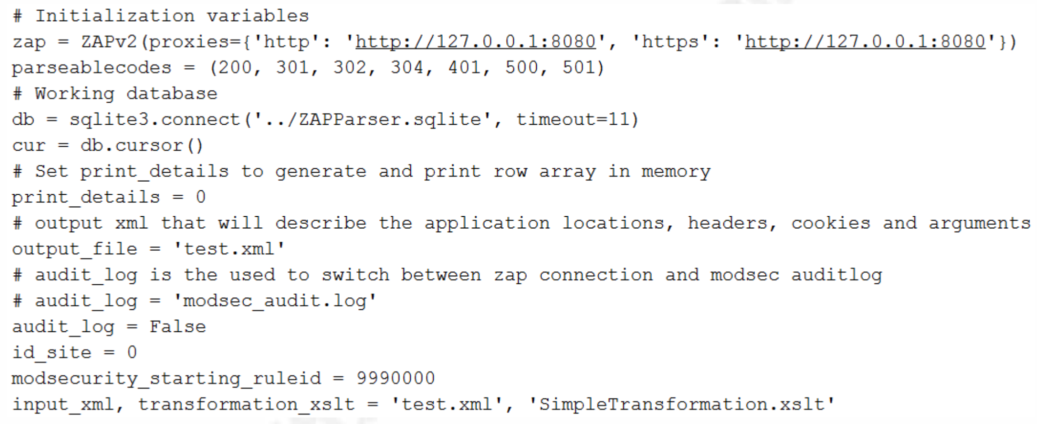
Figure 5 : ModSecurity settings for profile building

2.2.4. Configuring the ZAP application proxy for profiling

Using the OWASP Zed Attack Proxy (ZAP) to perform validation has multiple benefits: every request and response is stored into a persistent session file, vulnerabilities are identified in the application, the zap2modsec (Barnett R. , 2013) Perl script can be used to generate ModSecurity rules, etc. Before testing commences, the mode is set to Standard or Protected to prevent undesired automatic active testing. The WebAppProfiler script extracts the message data from zap using the python ZAP API to extract and aggregate the data while performing an initial profiling of the elements. This is accomplished by comparing different instances of the same element and matching the values against a set of basic regular expressions to guide profile generation.

2.2.5. Generating the profile

WebAppProfiler can be used to inspect the resulting ModSecurity audit log or ZAP using a persistent session once a test is completed. It parses the log and generates the entry points, headers, cookies and arguments list, which is later converted into an XML file that can be transformed into a set of ModSecurity rules. WebAppProfiler also exports the output to an SQLite database that can be used to narrow the results or discover additional patterns. WebAppProfiler requires the python-owasp-zap-v2.4, HTTP-parser and tabulate libraries, and its configuration is based on the variables listed in the initial section of the script.



```
# Initialization variables
zap = ZAPv2(proxies={'http': 'http://127.0.0.1:8080', 'https': 'http://127.0.0.1:8080'})
parseablecodes = (200, 301, 302, 304, 401, 500, 501)
# Working database
db = sqlite3.connect('../ZAPParser.sqlite', timeout=11)
cur = db.cursor()
# Set print_details to generate and print row array in memory
print_details = 0
# output xml that will describe the application locations, headers, cookies and arguments
output_file = 'test.xml'
# audit_log is the used to switch between zap connection and modsec auditlog
# audit_log = 'modsec_audit.log'
audit_log = False
id_site = 0
modsecurity_starting_ruleid = 9990000
input_xml, transformation_xslt = 'test.xml', 'SimpleTransformation.xslt'
```

Figure 6 : WebAppProfiler initialization variables

To generate a profile using ZAP, the browser or application must be configured to use it as a proxy and perform full-site navigation and testing. Some validation tools allow for the creation of automated test cases to replay on demand, which is useful for this type of task.

If a test has already been performed, the ModSecurity audit for the testing virtual host is retrieved, and the "`audit_log`" variable is adjusted to match the path and file name.

2.3. Generating the ModSecurity rules

Turning transaction logs into ModSecurity rules is quite straightforward. With minimal user input, the tool shows a summary of all the locations, arguments, cookies and headers, and all the transactions are stored in an SQLite database, which is used for aggregation after parsing. Basic regular expressions are evaluated against the contents of every element to find the best match, which is compared against every iteration of the same element and adjusted to be its most permissive value. For example, if an argument is first detected to contain all numeric digits, but later found to contain digits and letters, it will be switched to alphanumeric, and the longest length will be kept.

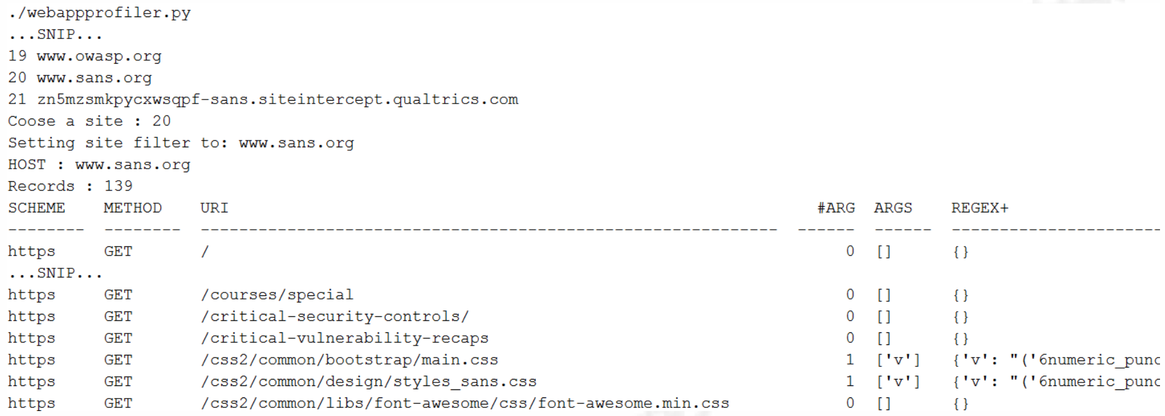
2.3.1. Using WebAppProfiler

By default, WebAppProfiler is configured to use ZAP running in localhost IP address 127.0.0.1 in port 8080 to retrieve messages with the 200, 301, 302, 304, 401, 500, and 501 response codes. The parseable codes are defined in the "`parseablecodes`" variable, the IP address and port of the ZAP proxy are specified in the "zap" variable, the name of the working database is defined under the "`db`" variable and the output file to store the XML profile is defined in the "`output_file`" variable. To switch from a ZAP connection to a ModSecurity audit log, the "`audit_log`" variable is set to the name of the log.

ModSecurity rule numbering is controlled by the variable "`modsecurity_starting_ruleid`", and its value is the first number the script uses to assign the identifier to every element. During the transformation phase, the script uses the "`input_xml`" and "`transformation_xslt`" variables.

The mandatory headers, parameters and cookies are dictionaries that can be customized with the name of the elements along with the "`False`" boolean value for optional elements, the "`True`" boolean value for mandatory elements or a mandatory "string" value. In addition, the cookies dictionary can accept "`SessionCookie`" string values. If there is a cookie match to the dictionary containing a SessionCookie value, it will be added to the additional session handling element and processing rules.

When WebAppProfiler starts in ZAP connected mode, it displays a list of available sites to choose from, whereas if it is configured in audit log mode, it starts processing immediately. After parsing the messages, it displays a summary of the parsed elements, including the scheme (http or https), the HTTP method used (GET, POST, etc.), the resource (URI) number or different arguments detected (#arg), the list or arguments (args), the best matching regular expression (regex+) and the possible list of characters that may be problematic (regex-). The same details are also displayed for headers (#head, head, hregex+, hregex-) and cookies (#coo, cook, cregex+, cregex-).



```
./webappprofiler.py
...SNIP...
19 www.owasp.org
20 www.sans.org
21 zn5mzsmkpycxwsqpf-sans.siteintercept.qualtrics.com
Coose a site : 20
Setting site filter to: www.sans.org
HOST : www.sans.org
Records : 139
SCHEME    METHOD    URI                                                             #ARG  ARGS    REGEX+
--------  --------  ------------------------------------------------------------  ------  ------  ----------------------
https     GET       /                                                                  0  []      {}
...SNIP...
https     GET       /courses/special                                                   0  []      {}
https     GET       /critical-security-controls/                                       0  []      {}
https     GET       /critical-vulnerability-recaps                                     0  []      {}
https     GET       /css2/common/bootstrap/main.css                                    1  ['v']   {'v': "('6numeric_punc
https     GET       /css2/common/design/styles_sans.css                                1  ['v']   {'v': "('6numeric_punc
https     GET       /css2/common/libs/font-awesome/css/font-awesome.min.css            0  []      {}
```

Figure 7 : WebAppProfiler aggregation summary

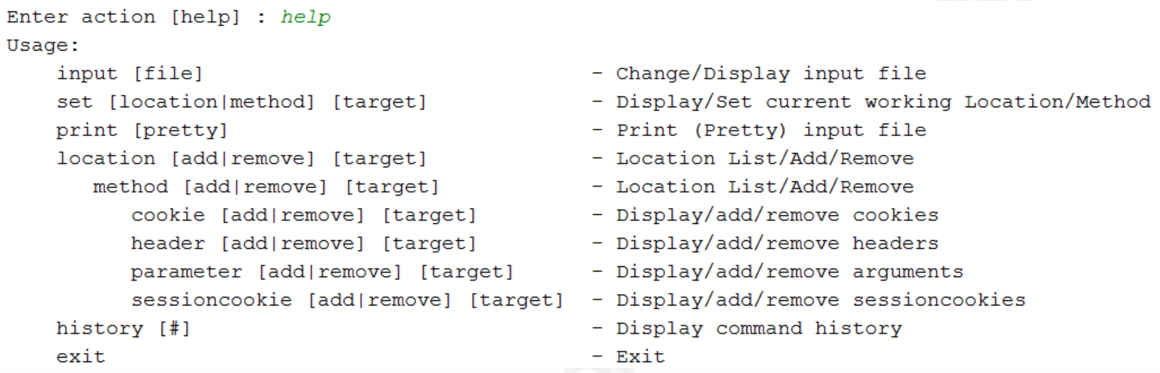
SQLite tables are built while parsing the messages for each of the different elements and are related by their resource identifier. After parsing, a profile is written in XML describing the site. Certain settings are assumed, which can be configured from the XML profile to subsequently add properties such as “required”, “rule score”, etc.

Once the profile is built, a simple transformation xslt file is used to generate ModSecurity compliant rules from the resulting XML profile. The independent simplerules transformation script is called as a module.

The process of generating the perfect rule set for a website’s needs may require performing multiple iterations and modifying the attributes of every element to obtain the ModSecurity rule with the desired actions for the requested elements.

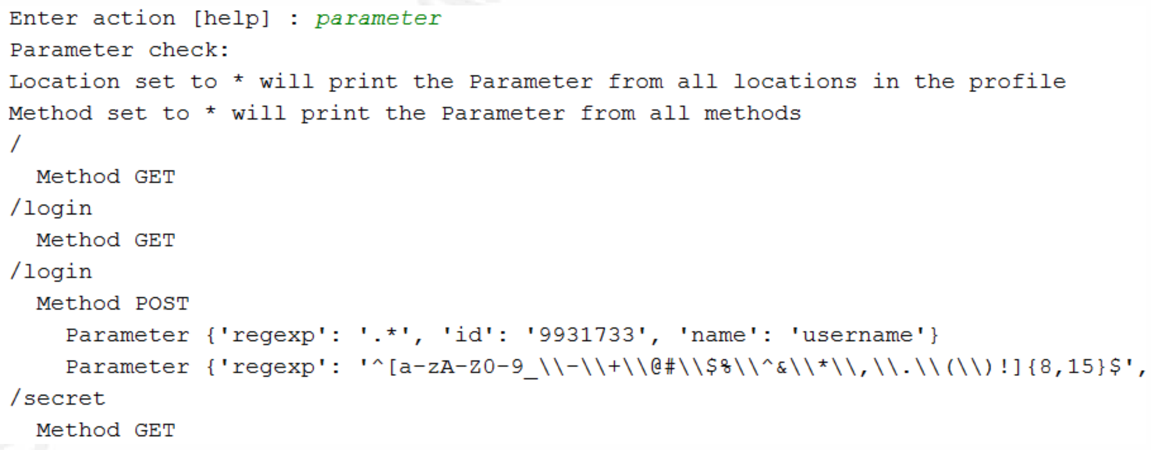
2.3.2. Modifying an XML profile

The profileeditor script provides several commands to modify the profile either in bulk or with respect to specific elements. The input XML profile can be changed or displayed with the input command, and the “`print`” command will display the current input file. If the “`pretty`” option is selected, the profile will be formatted with a hierarchical view. The location command adds or removes resource elements (URIs), and the set command configures the scope of the changes of location and methods. By default, both commands are set to affect every location and every available method. The method command adds or removes methods to the location in scope. The cookie, header, parameter and sessioncookie commands are linked to the location and methods in scope. If the location or method does not exist, nothing happens. If a method exists only in some locations, it affects only these locations.



```
Enter action [help] : help
Usage:
    input [file]                                  - Change/Display input file
    set [location|method] [target]                - Display/Set current working Location/Method
    print [pretty]                                - Print (Pretty) input file
    location [add|remove] [target]                - Location List/Add/Remove
       method [add|remove] [target]               - Location List/Add/Remove
          cookie [add|remove] [target]            - Display/add/remove cookies
          header [add|remove] [target]            - Display/add/remove headers
          parameter [add|remove] [target]         - Display/add/remove arguments
          sessioncookie [add|remove] [target]     - Display/add/remove sessioncookies
    history [#]                                   - Display command history
    exit                                          - Exit
```

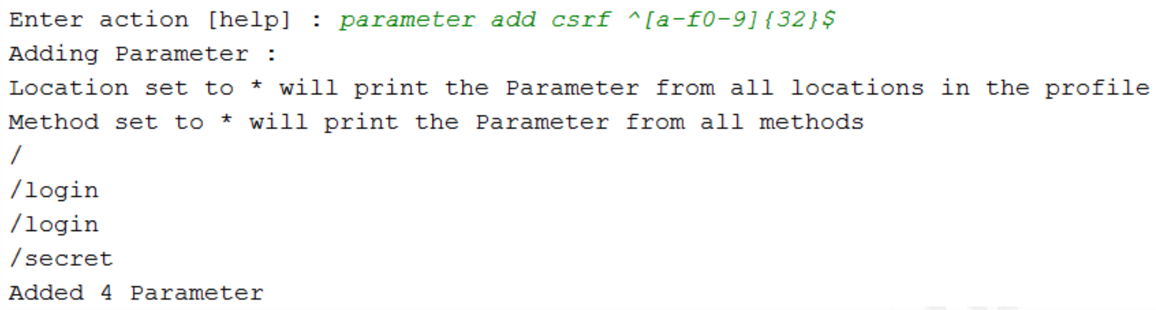
Figure 8 : profileeditor help screen



```
Enter action [help] : parameter
Parameter check:
Location set to * will print the Parameter from all locations in the profile
Method set to * will print the Parameter from all methods
/
  Method GET
/login
  Method GET
/login
  Method POST
    Parameter {'regexp': '.*', 'id': '9931733', 'name': 'username'}
    Parameter {'regexp': '^[a-zA-Z0-9_\\-\\+\\@#\\$%\\^&\\*\\,\\.\\(\\)!]{8,15}$',
/secret
  Method GET
```

Figure 9 : profileeditor parameter check

If the cookie, header, parameter and/or sessioncookie commands are called without arguments, an interactive mode is elicited to obtain the details of the element to add or remove. If the command is called with arguments, it will perform the action without asking for confirmation as long as the location is in scope and the method exists.



```
Enter action [help] : parameter add csrf ^[a-f0-9]{32}$
Adding Parameter :
Location set to * will print the Parameter from all locations in the profile
Method set to * will print the Parameter from all methods
/
/login
/login
/secret
Added 4 Parameter
```

Figure 10 : profileeditor adding parameters in bulk mode

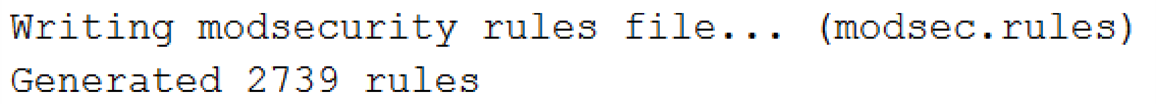
By default, location match blocks are used to contain the rules that are solely relevant to a particular resource. Each location is also split into multiple method blocks wherein the arguments, headers and cookies are checked. Alerts are triggered if there are missing elements or mismatches to the original regular expression and length.

To avoid processing unnecessary rules and variables, there are multiple "`skip`" actions in the rules to flag whether a check was performed. If this flag is missing at the end of the process, an alarm is triggered. The rule set adds session score counters to enable tracking across different transactions; however, the mechanism for initiating the session requires adding the '`SessionCookie`' attribute to acceptable cookies.

Risk ratings are not identical for all applications, and certainly not for each and every resource or element within an application; thus, for the sake of prioritizing alerts, scores are assigned to differentiate the most important sections from innocuous content, and the profile must be adjusted to have higher score values for sensitive elements or resources and low scores for the majority of the items. This allows for some alarms to be triggered without blocking the request. If a banned action is detected, it should have a high score so the transaction can be blocked.

ModSecurity supports multiple actions that may be added in the xslt file to match website policies and requirements. By default, the tool will not add any blocking action if there is no valid resource match, with the exception of a redirection action. In detection only mode, the rule engine will not perform redirection.

By default, many of the rules have audit log and log actions, which increase the amount of logging performed. For most elements, it is adequate to activate the log action only, removing the audit log and enabling it solely for sensitive operations.

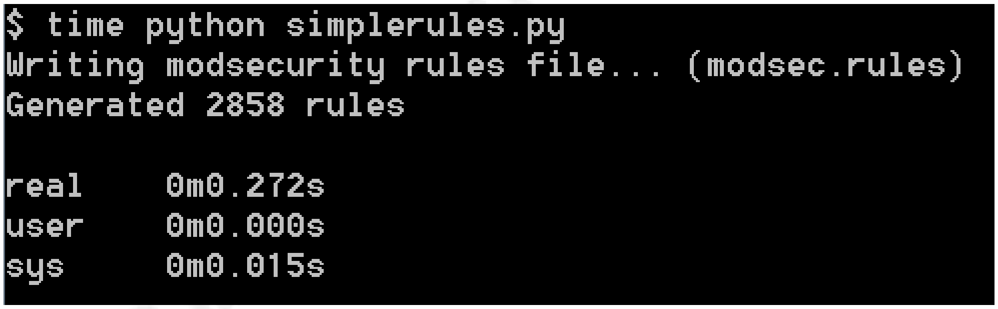


```
Writing modsecurity rules file... (modsec.rules)
Generated 2739 rules
```

Figure 11 : WebAppProfiler ModSecurity rule generation

2.3.3. Transforming from conceptual language to actionable rules

Once the conceptual model is fully described in the XML profile, the xslt transformation finishes in a matter of seconds. Before proceeding, the transformation adjusts attributes such as scores, values and session cookie identification. The simplerules python script takes the XML as input and uses the SimpleTransformation.xslt file to generate ModSecurity rules that are ready to be loaded into the server.

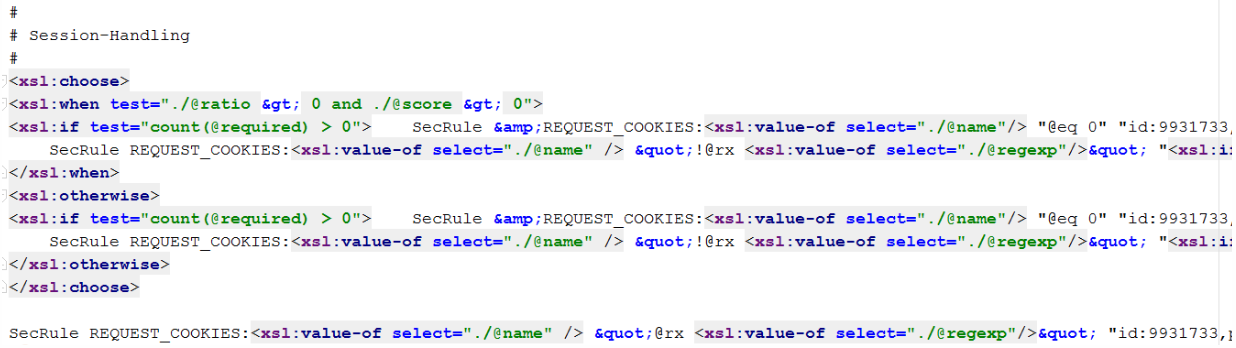


```
$ time python simplerules.py
Writing modsecurity rules file... (modsec.rules)
Generated 2858 rules

real    0m0.272s
user    0m0.000s
sys     0m0.015s
```

Figure 12 : Running the simplerules transformation script

The automatically generated rules may be modified independently in the resulting file, but for changes that apply to all the rules, it is advisable to modify the xslt file.

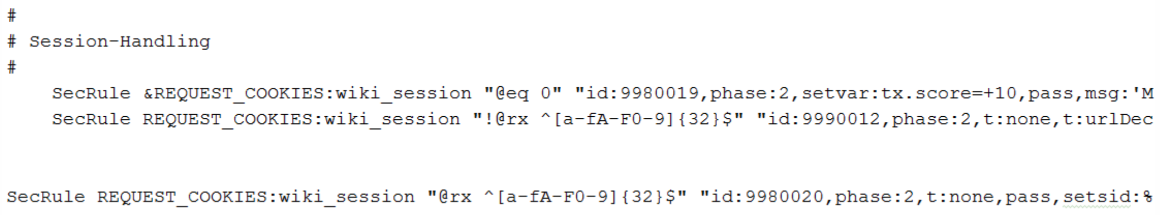


```
#
# Session-Handling
#
<xsl:choose>
<xsl:when test="./@ratio &gt; 0 and ./@score &gt; 0">
<xsl:if test="count(@required) > 0">    SecRule &amp;REQUEST_COOKIES:<xsl:value-of select="./@name"/> "@eq 0" "id:9931733,
    SecRule REQUEST_COOKIES:<xsl:value-of select="./@name" /> &quot;!@rx <xsl:value-of select="./@regexp"/>&quot; "<xsl:i
</xsl:when>
<xsl:otherwise>
<xsl:if test="count(@required) > 0">    SecRule &amp;REQUEST_COOKIES:<xsl:value-of select="./@name"/> "@eq 0" "id:9931733,
    SecRule REQUEST_COOKIES:<xsl:value-of select="./@name" /> &quot;!@rx <xsl:value-of select="./@regexp"/>&quot; "<xsl:i
</xsl:otherwise>
</xsl:choose>

SecRule REQUEST_COOKIES:<xsl:value-of select="./@name" /> &quot;@rx <xsl:value-of select="./@regexp"/>&quot; "id:9931733,
```

Figure 13 : Session handling block in the xslt file

The rule id must be unique, so the xslt takes the id property of the element, if available, and applies the generic id `9931733`, and the simplerules script replaces it while creating the rules. It will not check for duplicates; thus, any manual addition of elements must use the generic id `9931733` so they can be replaced during the transformation phase.



```
#
# Session-Handling
#
    SecRule &REQUEST_COOKIES:wiki_session "@eq 0" "id:9980019,phase:2,setvar:tx.score=+10,pass,msg:'M
    SecRule REQUEST_COOKIES:wiki_session "!@rx ^[a-fA-F0-9]{32}$" "id:9990012,phase:2,t:none,t:urlDec


SecRule REQUEST_COOKIES:wiki_session "@rx ^[a-fA-F0-9]{32}$" "id:9980020,phase:2,t:none,pass,setsid:%
```

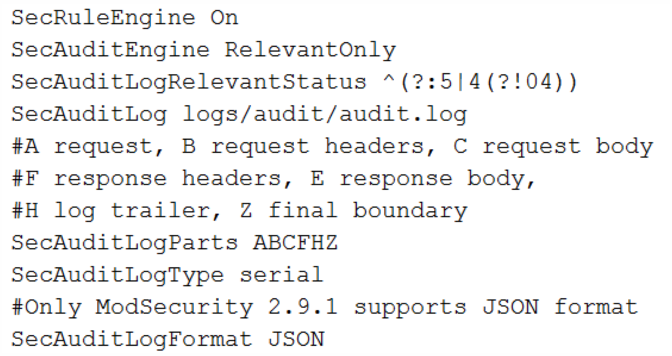
Figure 14 : ModSecurity rules produced by the xslt session handling block

3. Deploying the profile

The ModSecurity rules produced by the simplerules script can be imported directly into the virtual host using the Apache httpd server *Include* or *IncludeOptional* directives, and the files should be located in different directories depending on the operating system.

3.1. Preparing the production environment

In the production environment, ModSecurity setup is where all the previously generated rules are installed for improving the intrusion detection capabilities of the site and protecting the application from undesired anomalies. It is highly advisable to use a complementary rule set for detecting known attacks such as SQL injection and cross-site scripting. The volume of requests to internet servers, including malicious requests, may be very high. These requests produce a massive volume of logs to be rotated, compressed and archived. Frequently, the ModSecurity audit engine settings only log transactions with response status codes in the 400-599 range, with the exception of 404 (not found), and the body of the response is rarely logged (part E).



```
SecRuleEngine On
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus ^(?:5|4(?!04))
SecAuditLog logs/audit/audit.log
#A request, B request headers, C request body
#F response headers, E response body,
#H log trailer, Z final boundary
SecAuditLogParts ABCFHZ
SecAuditLogType serial
#Only ModSecurity 2.9.1 supports JSON format
SecAuditLogFormat JSON
```

Figure 15 : ModSecurity settings in a production server

3.2. Apache, ModSecurity, OWASP CRS

Apache is currently the most commonly used web server, with 34.48% (Netcraft, 2016) of internet sites using it. Any server connected to the internet must be hardened (CIS, 2016), and its logs must be frequently monitored. Running a web application firewall to protect internet servers from attacks is useful, but a WAF is only as good as the rules it uses. The combination of OWASP CRS and the newly created profile complement each other to improve an application's security. The CRS detects known anomalies, while the profile highlights any unusual value in the transaction and indicates whether new elements have been discovered

One shortcoming of a negative security approach is that most of the rules are static and require updates due to the endless race of evolving hacking tools, bypasses, vulnerabilities and techniques.

Because a WAF typically gives the best visibility into web application traffic, positive security measures in conjunction with a negative security rule alert feed into a security incident and event management (SIEM) system can enhance monitoring and detection capabilities (Pubal, 2015).

3.3. Basic logs available and their purpose

Apache provides two major log types, one for including all the transactions processed by the server and another for errors. The format of the log may be modified (The Apache Software Foundation, 2016, log formats) to have multiple fields, which are limited to the characteristics of the transaction, the query string, protocols, methods, ciphers and the original IP address. As described by the httpd documentation, the error log is “the most important log file. This is the place where Apache httpd will send diagnostic information and record any errors that it encounters in processing requests” (The Apache Software Foundation, 2016, Apache error logs). The error log is also updated by ModSecurity if the log action is present in the rule.

ModSecurity also has logs, and it will write the request and response headers and body in addition to all the ModSecurity output when a rule is triggered as long as the audit log rule is present in the rule, depending on the configuration.

3.3.1. Logging considerations

A log’s growth rate and its frequency of review directly affect its retention and rotation. A higher log volume requires more frequent rotation, and if there is a central log collection and correlation system available, it is likely to offer a mechanism for log secure transportation, including encryption and compression. In addition, several file elements are frequently repeated, making log files highly compressible in general.

The format (The Apache Software Foundation, 2016, Apache mod_log_config) of the log is important, as any changes will affect the parsing tools and configurations of scripts and SIEM; thus, it is important to set it up properly from the beginning. The default format is often sufficient, but a few additions can be useful, such as replacing the host name (%h) by the client IP address (%a) to prevent DNS requests. If the server receives requests from reverse proxies or load balancers, the client IP will provide either the balancer or proxy; thus, if they are trusted, it is advisable to include the underlying peer IP (%{c}a), which uses the information provided in the headers to get the original IP. As any header can be manipulated by the client, only headers coming directly from trusted servers are trusted. If multiple servers are being managed, it is also wise to have the server address (%A) and the transaction unique id (SpiderLabs, ModSecurity reference manual, 2016) (%{UNIQUE_ID}e or %L) (The Apache Software Foundation, 2016, Apache custom log formats) to ease correspondence with the ModSecurity audit and server error log.

In Apache 2.4, the error log format is customizable. The log ID format (%L) is a must-add because it produces a unique id for a connection or request. Because

ModSecurity requires mod_unique_id (The Apache Software Foundation, 2016, Apache module mod_unique_id) to be loaded, all the log files use the same identifier per transaction.

Logs should be reviewed often, and an automated process to normalize and aggregate the events may be mandatory for managing multiple servers or high traffic sites. To this end, there are freely available products from elastic (Elastic, 2016) and jwall (Bockermann, jwall AuditConsole, 2015) AuditConsole (Pubal, 2015) that simplify the log review process, and "Your best choice for a log centralization and GUI tool is AuditConsole" (Ristic, 2012).

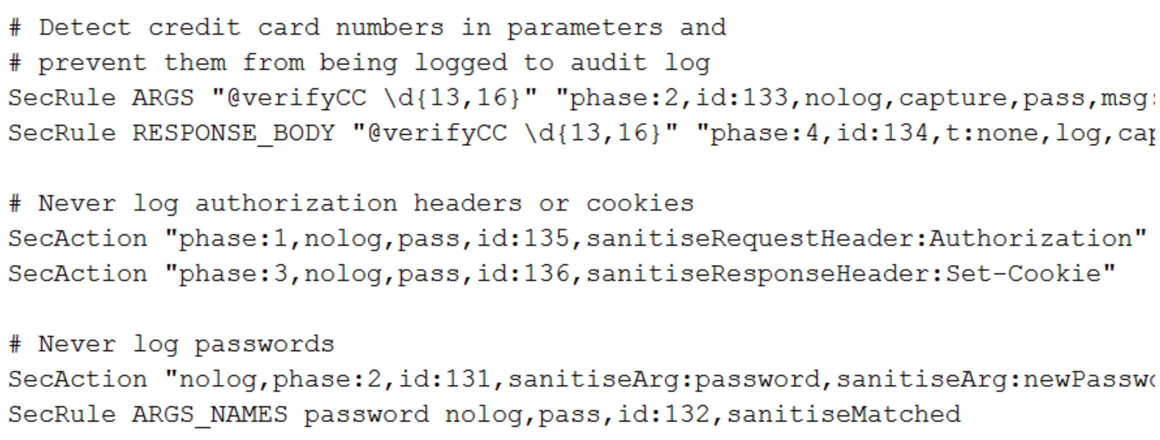
3.3.2. Log protection and sanitization

Many companies and countries are subject to privacy and sensitive data protection laws and regulations, some of which impose huge fines for compromised data. One side effect of the ModSecurity audit log is that it can dump the entire request and response into the audit log, potentially creating a large repository of data including card holder data, personal data, health care data and user credentials.

The log files and the containing directory must be secured, restricting the access as much as possible, following Apache security recommendations (The Apache Software Foundation, 2016, Apache security tips). Whenever possible, sensitive data should be removed from the logs, and it is highly advisable to avoid writing such data to the logs in the first place.

The audit log may be full of sensitive data, often in the request or response body section (C, E and I parts). In such cases, the SecAuditLogParts (SpiderLabs, ModSecurity reference manual, 2016) directive should not list the affected log areas. It is important to remember that some rules may dynamically change these settings, i.e., certain rules in the OWASP CRS have the "`ctl:auditLogParts=+E`" action, which puts the response body in the transaction's audit log.

ModSecurity has multiple actions to sanitize (SpiderLabs, ModSecurity reference manual, 2016) log elements, including "`sanitiseArg`" for arguments, "`sanitiseMatched`" for rule target matches, "`sanitiseMatchedBytes`" for only the rule matching bytes and "`sanitiseRequestHeader`" and "`sanitiseResponseHeader`" for HTTP headers.



```
# Detect credit card numbers in parameters and
# prevent them from being logged to audit log
SecRule ARGS "@verifyCC \d{13,16}" "phase:2,id:133,nolog,capture,pass,msg:
SecRule RESPONSE_BODY "@verifyCC \d{13,16}" "phase:4,id:134,t:none,log,cap

# Never log authorization headers or cookies
SecAction "phase:1,nolog,pass,id:135,sanitiseRequestHeader:Authorization"
SecAction "phase:3,nolog,pass,id:136,sanitiseResponseHeader:Set-Cookie"

# Never log passwords
SecAction "nolog,phase:2,id:131,sanitiseArg:password,sanitiseArg:newPassw
SecRule ARGS_NAMES password nolog,pass,id:132,sanitiseMatched
```

Figure 16 : ModSecurity log sanitization rules (SpiderLabs, 2016)

Apache logs can also create leaks of sensitive data. This generally happens when sensitive data are sent in the query string. A special condition can be added to create an environmental variable whenever this is detected to prevent the transaction from being logged and log it without the offending item.

3.4. Maintaining the profile

The simplerules generated ModSecurity rules have multiple events that must be reviewed and used as feedback to update or improve the profile. Depending on the extent of validation used, the number of items to modify will change over time. It is thus necessary to identify all the relevant events that signify whether a request contains anomalies such as missing elements or elements with values that do not match the validation regular expression.

3.4.1. Identify significant events

In positive security model rules, the events generated are related to missing elements such as cookies, headers, session cookies and parameters, resources that are not listed in the permitted locations and the use of methods that are not allowed in a given location.

ModSecurity will write an event in the audit log as per the configuration of the xslt transformation file.

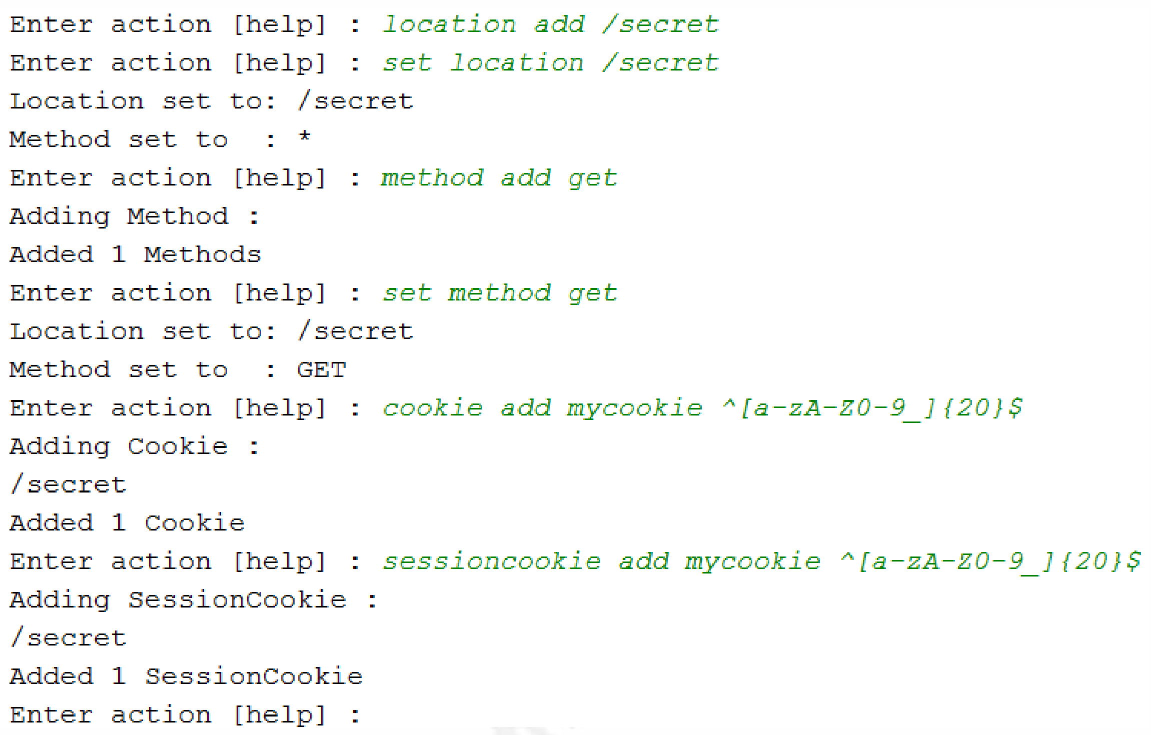
An ‘Invalid URL requested’ event is triggered when a requested URI is not listed in the profile and the xslt file contains two variants of this event. The default rule logs the requested URI and the parameters but still allows the request. In addition, the other rule redirects the request to a non-existent page, and the client receives an error message.

'Missing required parameter/cookie/header/sessioncookie’ events are triggered when these elements have the “`required`” attribute and are not present in the request.

The elements are compared against a regular expression; if a value does not match the expression, the transaction risk score is increased, and a ‘transaction-score’ event is generated, which includes a summarization, after checking all the elements in the transaction. If session cookies are declared, collection is initiated, and the session-score is tracked across all the transactions and connections containing the same session-cookie.

3.4.2. Analyzing the event and identifying the related events

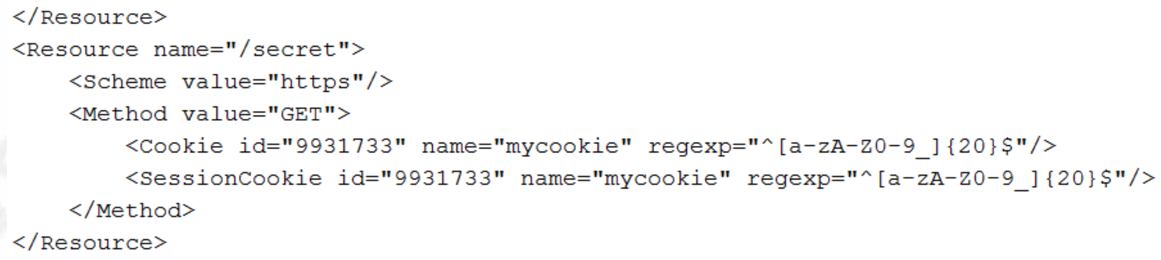
Invalid URI identification is achieved by searching for ‘Invalid URL requested’ events in the audit log. These events also include the requested resource and the arguments of the request. If the redirection rule is enabled in the xsl file, ModSecurity will reject any requests not listed in the profile by default. Otherwise, it triggers the alert without affecting the request in any way. For all allowable resources, new locations must be added into the profile, which must include the allowed method, cookies, headers, parameters and session cookies.



```
Enter action [help] : location add /secret
Enter action [help] : set location /secret
Location set to: /secret
Method set to  : *
Enter action [help] : method add get
Adding Method :
Added 1 Methods
Enter action [help] : set method get
Location set to: /secret
Method set to  : GET
Enter action [help] : cookie add mycookie ^[a-zA-Z0-9_]{20}$
Adding Cookie :
/secret
Added 1 Cookie
Enter action [help] : sessioncookie add mycookie ^[a-zA-Z0-9_]{20}$
Adding SessionCookie :
/secret
Added 1 SessionCookie
Enter action [help] :
```

Figure 17 : profileeditor adding a location

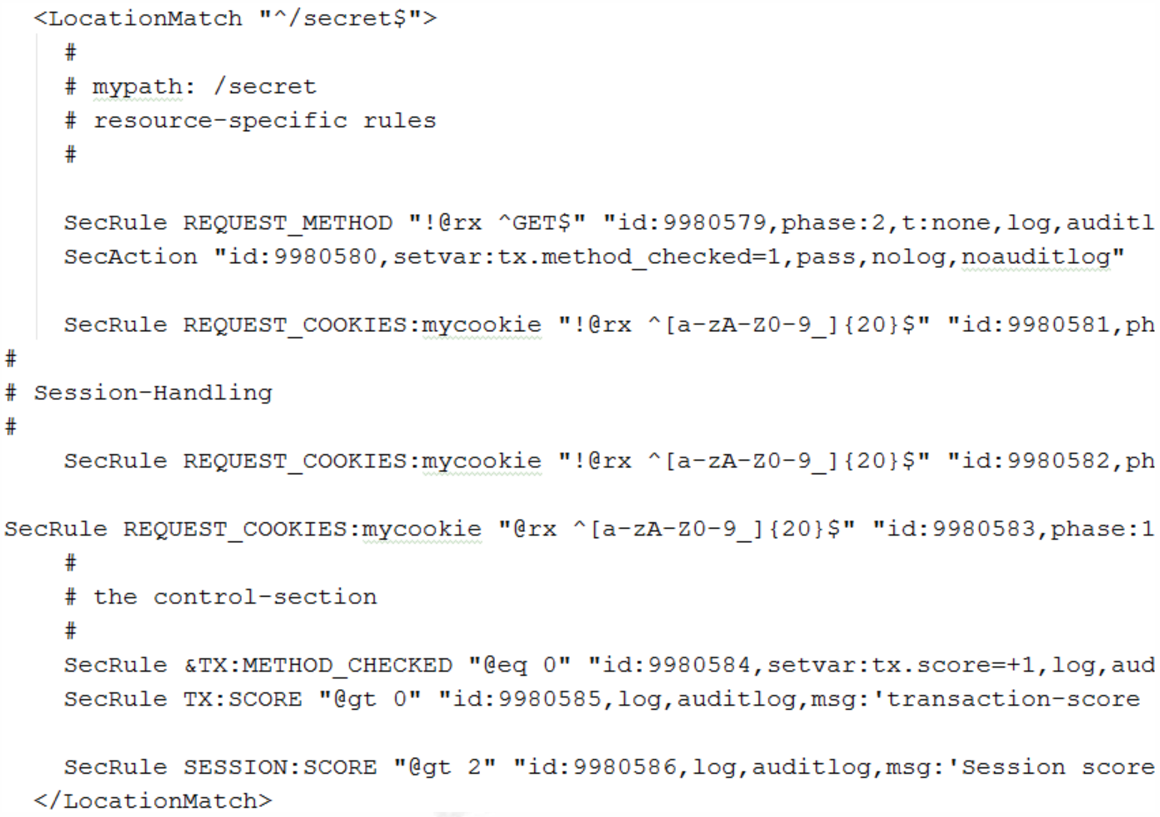
The XML profile changes are performed at the moment the commands are executed in profileeditor. After the changes are completed, the "`print pretty`" command can be used to visualize the entire file. New location additions are located at the end of the file, and all other changes are located at the end of their respective parents.



```
</Resource>
<Resource name="/secret">
    <Scheme value="https"/>
    <Method value="GET">
        <Cookie id="9931733" name="mycookie" regexp="^[a-zA-Z0-9_]{20}$"/>
        <SessionCookie id="9931733" name="mycookie" regexp="^[a-zA-Z0-9_]{20}$"/>
    </Method>
</Resource>
```

Figure 18 : XML elements added to create a location

The more strict and accurate a regular expression is, the better it works for the positive security model, and generic expressions such as "`.*`" must be avoided as much as possible. Profileeditor generates the entire rule set in seconds. For version control, the previous rule set can be renamed and then replaced with the new file, and a server reload enables the new configuration.



```
  <LocationMatch "^/secret$">
    #
    # mypath: /secret
    # resource-specific rules
    #

    SecRule REQUEST_METHOD "!@rx ^GET$" "id:9980579,phase:2,t:none,log,auditl
    SecAction "id:9980580,setvar:tx.method_checked=1,pass,nolog,noauditlog"

    SecRule REQUEST_COOKIES:mycookie "!@rx ^[a-zA-Z0-9_]{20}$" "id:9980581,ph
#
# Session-Handling
#
    SecRule REQUEST_COOKIES:mycookie "!@rx ^[a-zA-Z0-9_]{20}$" "id:9980582,ph

SecRule REQUEST_COOKIES:mycookie "@rx ^[a-zA-Z0-9_]{20}$" "id:9980583,phase:1
    #
    # the control-section
    #
    SecRule &TX:METHOD_CHECKED "@eq 0" "id:9980584,setvar:tx.score=+1,log,aud
    SecRule TX:SCORE "@gt 0" "id:9980585,log,auditlog,msg:'transaction-score

    SecRule SESSION:SCORE "@gt 2" "id:9980586,log,auditlog,msg:'Session score
  </LocationMatch>
```

Figure 19 : ModSecurity rules of the added location

To identify the elements that are not present in the allowed list, the ‘Found new’ events can be extracted 1) from the logs, using SIEM search capabilities or 2) from the command line, using tools such as grep and egrep (see examples below) to get the triggered rule id, the new element found and the location where it was found. Some elements cause much higher levels of log noise than others; thus, verifying anomalies in reverse order is a good idea because fixing the most recurrent event will result in the largest log volume reduction.

```
$ egrep -o "\[id.+Found new parameter '/.+/' in [^\"]+"
modsec_audit.log |sed -rn "s,\[id (.+)\].+Found.+'/(.+)/' in
(.+),\1 \2 \3,p" |sort |uniq -c |sort -nr
```

The values contained in these elements can then be extracted from the log to identify the different types of characters or values used.

```
$ egrep -o "argument=.+" modsec_audit.log |sort |uniq
```

The value list can be analyzed, and a regular expression to match the values cam be made as restrictive as possible, i.e., ^[a-zA-Z]{5,10}$; if there are only a few different values, a list can be used instead, i.e., ^(?:apples|oranges|bananas)$, and profileeditor can be used to modify the XML profile, followed by simplerules to replace the current set of rules with a new set of rules.

The approach is identical for adjusting cookies headers and session cookies: identify new elements from the events, search for the values in the audit log by searching for the rule id, verify whether the element is normal for the application and should be allowed to access the particular resource and method, get the different values of every instance of the element, write a filter that will solely match these values, update the XML profile accordingly, generate a new set of rules based on the new profile, test and deploy.

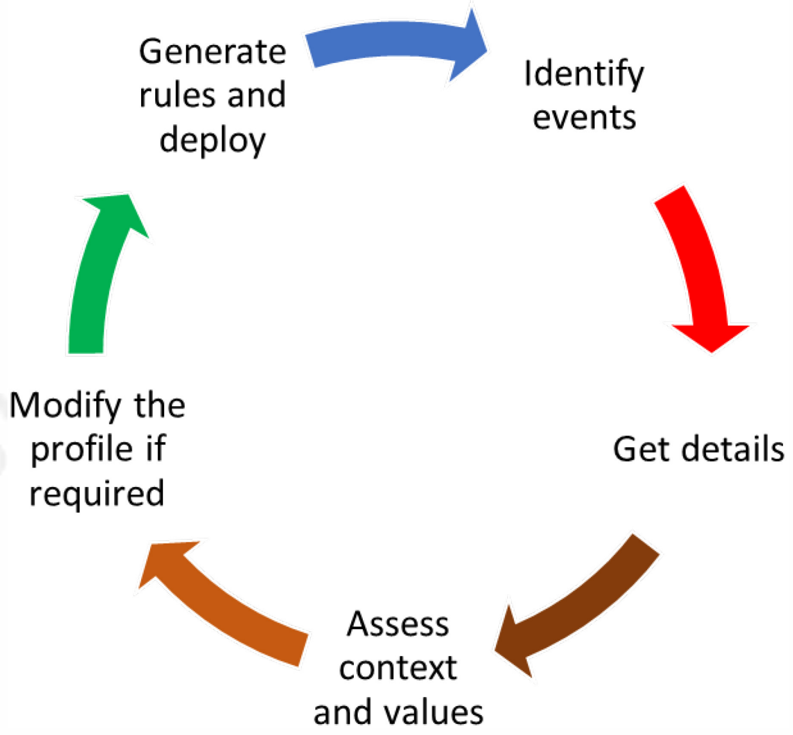

Generate rules and deploy
Identify events
Get details
Assess context and values
Modify the profile if required

Figure 20 : Maintaining the profile process

4. Conclusion

Not all web application firewalls are identical. Some favor ease of integration over security, and most out-of-the-box products are capable of protecting web applications using a negative security model to identify potential anomalies. A negative security model requires continuous maintenance and evolution to stay on top of attacks, vulnerabilities and bypasses.

A positive security model can be used to generate ModSecurity rules to allow recognized and valid values to pass through the firewall and generate an alert or block access if the received values differ from the expected values.

Using a combination of negative and positive models offers improved application protection and detection capabilities. If it is impossible to use a positive security filter, the negative model prevents known faulty patterns from being entered. Simultaneously, if the negative filter ignores a new type of filter bypass or attack, the positive filter limits the amount and type of receivable input.

Automation allows for the quick generation of a basic blanket policy, and its effectiveness can be significantly enhanced by making the filters more specific to the elements and context by accepting only a few known values and rejecting everything else.

As with any other security control method, intrusion detection in web application firewalls must be cyclic and encompass log review, the identification of anomalies and false positives, profile improvements, rule set regeneration, testing and deployment.

5. References

Auger, R., Barnett, R., Cano, C., Chuvakin, A., Estrade, M., Ristic, I., et al. (2006, January). *Web Application Firewall Evaluation Criteria.* Retrieved from Web Application Security Consortium Project: http://projects.webappsec.org/f/wasc-wafec-v1.0.pdf

Barnett, R. (2013). *zap2modsec*. Retrieved from SpiderLabs github: https://github.com/SpiderLabs/owasp-modsecurity-crs/blob/master/util/virtual-patching/zap2modsec.pl

Barnett, R. C. (2012). *Web application defender's cookbook: Battling hackers and protecting users.* United States: Wiley, John & Sons.

Berners-Lee, T., Fielding, R., Masinter, L., Day Software, Adobe Systems, & W3C/MIT. (2005, January). *RFC3986 Uniform Resource Identifier (URI): Generic Syntax*. Retrieved from IETF: https://tools.ietf.org/html/rfc3986

Bockermann, C. (2007, April 05). *A Meta-Language for Web Application Profiles.* Retrieved from jwall java based web security tools: http://www.jwall.org/web/profile/meta-language.pdf

Bockermann, C. (2007, April 05). *jwall web application profiler*. Retrieved from jwall java based web security tools: https://www.jwall.org/web/profiler/

Bockermann, C. (2007, April 05). *Web Application Profiles.* Retrieved from jwall java based web security tools: http://jwall.org/web/policy/lang.jsp

Bockermann, C. (2007, April 05). *Web Profile Editor.* Retrieved from jwall java based web security tools: http://www.jwall.org/web/profile/editor.jsp

Bockermann, C. (2015, 01 15). *jwall AuditConsole*. Retrieved from jwall java based web security tools: http://jwall.org/web/audit/console/index.jsp

Crockford, D. (1999). *JavaScript Object Notation JSON*. Retrieved from Introducing JSON: http://www.json.org/

Crockford., D. (1999). *Introducing JSON*. Retrieved from JSON JavaScript Object Notation: http://www.json.org/

Elastic. (2016). *Logstash Elastic*. Retrieved from Elastic: https://www.elastic.co/products/logstash

The Python Software Foundation. (2016). *Python 2.7*. Retrieved from Python: https://www.python.org/

The Python Software Foundation. (2016). *Python HTTP-parser*. Retrieved from Python Software: https://pypi.python.org/pypi/http-parser/

The Python Software Foundation. (2016). *Python OWASP ZAP API*. Retrieved from Python: https://pypi.python.org/pypi/python-owasp-zap-v2.4

The Python Software Foundation. (2016). *Python tabulate*. Retrieved from Python Software: https://pypi.python.org/pypi/tabulate/

The Apache Software Foundation. (2016). *Apache custom log formats*. Retrieved from Apache: https://httpd.apache.org/docs/2.4/en/mod/mod_log_config.html#formats

The Apache Software Foundation. (2016). *Apache error log*. Retrieved from Apache: https://httpd.apache.org/docs/2.4/en/logs.html#errorlog

The Apache Software Foundation. (2016). *Apache HTTP Server Project*. Retrieved from Apache: http://httpd.apache.org/

The Apache Software Foundation. (2016). *Apache mod_log_config*. Retrieved from Apache: https://httpd.apache.org/docs/2.4/en/mod/mod_log_config.html

The Apache Software Foundation. (2016). *Apache module mod_unique_id*. Retrieved from Apache: http://httpd.apache.org/docs/2.4/mod/mod_unique_id.html

The Apache Software Foundation. (2016). *Apache security tips*. Retrieved from Apache: http://httpd.apache.org/docs/2.4/misc/security_tips.html

The Apache Software Foundation. (2016). *Apache Virtual Host documentation*. Retrieved from Apache: http://httpd.apache.org/docs/2.4/vhosts/

Leos Rivas, M. (2016, August). *WebAppProfiler*. Retrieved from Spartantri github: https://github.com/spartantri/webappprofiler

Meyer, R. (2008, January 26). *Detecting attacks on Web applications from log files.* Retrieved from SANS reading room: https://www.sans.org/reading-room/whitepapers/logging/detecting-attacks-web-applications-log-files-2074

Muntner , A. (2010). *Official FuzzDB project repository*. Retrieved from FuzzDB: https://github.com/fuzzdb-project/fuzzdb

Netcraft. (2016, July). *July 2016 Web Server*. Retrieved from Netcraft: https://news.netcraft.com/archives/2016/07/19/july-2016-web-server-survey.html

Nottingham, M. (2014, July). *RFC7320 URI Design and Ownership*. Retrieved from IETF: https://tools.ietf.org/html/rfc7320

OWASP. (2016). *OWASP ModSecurity Core Rule Set Project*. Retrieved from OWASP CRS: https://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project

OWASP. (2016). *OWASP Zed Attack Proxy*. Retrieved from OWASP ZAP: https://www.owasp.org/index.php/ZAP

Pubal, J. (2015, March 13). *Web Application Firewalls Enterprise Techniques.* Retrieved from SANS reading room: https://www.sans.org/reading-room/whitepapers/application/web-application-firewalls-35817

Ristic, I. &. (2008, August). *Enough with default allow.* Retrieved from ModSecurity: http://blog.modsecurity.org/files/Breach_Security_Labs-Enough_with_Default_Allow.pdf

Ristic, I. &. (2008, June 30). *No More Signatures: Defending Web Applications from 0-day attacks.* Retrieved from BlackHat: https://www.blackhat.com/presentations/bh-usa-08/Ristic_Shezaf/BH_US_08_No_More_Signatures_Ivan_Ristic_Ofer_Shezaf_Wp.pdf

Ristic, I. (2012). *ModSecurity handbook: The complete guide to the popular open source web application Firewall.* London, United Kingdom: Feisty Duck.

Security, C. f. (2016). *CIS security benchmarks*. Retrieved from the Center for Internet Security: https://benchmarks.cisecurity.org/downloads/benchmarks/

SpiderLabs, T. (2016). *ModSecurity Open Source Web Application Firewall*. Retrieved from ModSecurity: http://modsecurity.org/

SpiderLabs, T. (2016). *ModSecurity reference manual*. Retrieved from ModSecurity: https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual

W3C. (1999). *XSLT Namespace*. Retrieved from World Wide Web Consortium (W3C): http://www.w3.org/1999/XSL/Transform