SECURING APACHE : mod_security

Right from Part 1 of this series, we've covered the major types of attacks being done on Web applications — and their security solutions. In this article, I will reveal the tremendous capabilities of the Apache mod_security module, covering just a small part of what it can do.

From the development perspective, implementing security against the many attacks on Web apps doesn't just require extra coding and stronger validation, but often also results in complex and messy code, which may sometimes cause yet another security loophole.

Security is often compared to a football game, where success requires the defense to quickly adapt, outrun, and outplay the attackers. Such a dynamic defense cannot properly survive in complex and messy code. Here, Web application firewalls come to the rescue — and what else is better than mod_security.
It is designed as an Apache module that adds intrusion-detection and prevention features to the Web server. In principle, it's similar to an IDS that analyses network traffic, but it works at the HTTP level, and really well, at that. This allows you to do things that are difficult in a classic IDS. This difference will become clearer when we examine several examples. The attack prevention feature stands between the client and server; if it finds a malicious payload, it can reject the request, performing any one of a number of built-in actions.

Some of the features of mod_security are audit logging, access to any part of the request (including the body) and the response, a flexible regular expression-based rule engine, file-upload interception, real-time validation and also buffer-overflow protection.

How mod_security works

The module's functionality is divided into four main areas:

1. Parsing: Security-conscientious parsers extract bits of each request and/or response and store them for use in the rules.
2. Buffering: In a typical installation, both request and response bodies will be buffered so that the module usually sees complete requests (before they are passed to the application for processing), and complete responses (before they are sent to clients). Buffering is important, because it is the only way to provide reliable blocking.
3. Logging: Allows you to record complete HTTP traffic, logging all response/request headers and bodies.
4. Rule engine: Works on the data from the other components, to assess the transaction and take action, as necessary.

Deployment architectures

mod_security can be deployed in two modes:

Embedded mode: Just add mod_security as a module into your Apache Web server. However, in this mode, it is not able to inspect the content of server headers.

Network gateway: In this mode (recommended since all the Web traffic goes through the proxy),mod_security is installed as a reverse proxy (see Figure 1). If you are using this mode, ensure you also add mod_proxy and mod_proxy_http in Apache. This gives a single point to monitor, higher speed, high anonymity of the internal network, and mod_securitycan inspect the server header of the backend database.

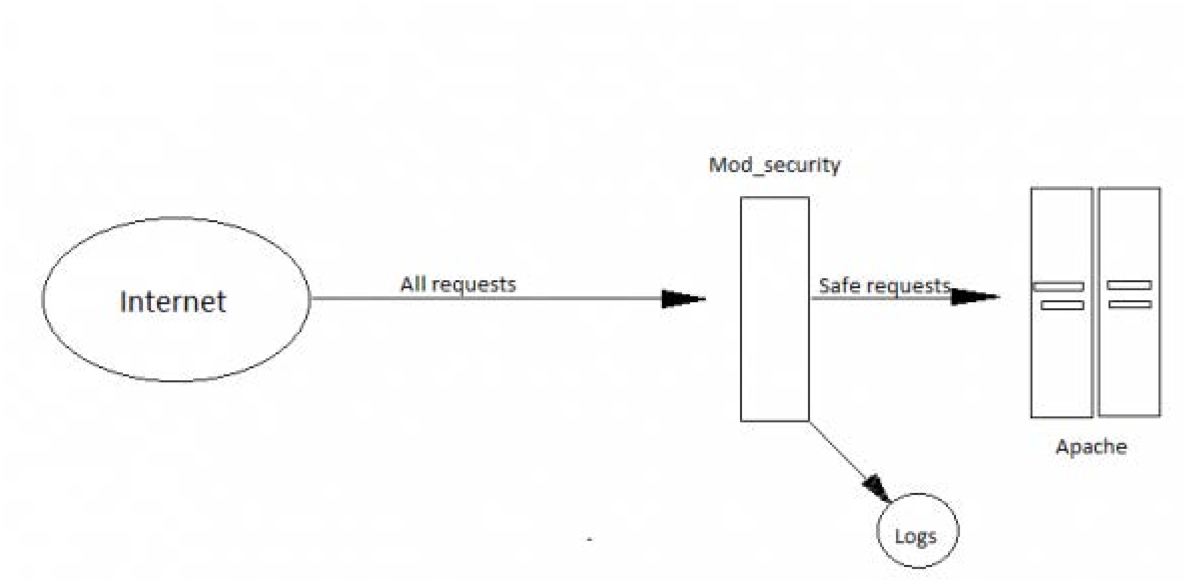

Figure 1: 'mod_security' as reverse proxy

Installation

Basically, mod_security works on configuration and rules. The configuration instructs it how to process the data it sees; the rules decide what to do with the processed data.

The configuration directives can be directly added to httpd.conf, but to avoid cluttering that, include a separate modsecurity.conf with this line in httpd.conf:

Include conf/modsecurity.conf

Installation is not straightforward, and also depends on the OS and Apache version you are using. But the slight trouble is negligible in comparison to the services mod_security offers. For detailed installation steps, refer to the documentation on modsecurity.org. A Windows version is also available, which has other customised versions of Apache and Apache modules.

Configuration

The basic modsecurity.conf looks like the following code:

```
<IfModule mod_security.c>
# Turn the filtering engine On or Off
SecFilterEngine On
# The audit engine works independently and can be turned
# On or Off on a per-server or per-directory basis
SecAuditEngine RelevantOnly
# Make sure that URL encoding is valid
SecFilterCheckURLEncoding On
# Unicode encoding check
SecFilterCheckUnicodeEncoding On
# Only allow bytes from this range
SecFilterForceByteRange 1 255
# Cookie format checks.
SecFilterCheckCookieFormat On
# The name of the audit log file
SecAuditLog logs/audit_log
# Should mod_security inspect POST payloads
SecFilterScanPOST On
# Default action set
SecFilterDefaultAction "deny,log,status:500"
</IfModule>
```

Now, let’s look at some basic configuration directives:

♣ SecFilterEngine: When set to On (that is, SecFilterEngine On), it starts monitoring requests. It is Off (disabled) by default.

- SecFilterScanPOST: When On, enables scanning the request body/POST payload.
- SecFilterScanOutput: When On, enables scanning the response body also.

Similarly, to check URL encoding, you can use SecFilterCheckURLEncoding; to control request body buffering, use SecRequestBodyAccess; to control what happens once the response body limit is reached, use SecResponseBodyLimitAction; and to specify the response body buffering limit, use SecResponseBodyLimit.

The full list of configuration directives, their usage and syntax is at available on modsecurity.org.

Rules — the basics

The mod_security rule engine is where gathered data is checked for any malicious or particular content. Rules are directives in the configuration file that decide what to do with the data parsed by the configuration directives. The rule language is a vast topic; we'll only discuss basic rule-writing syntax, and rule directives to secure Web applications from all the attacks we've discussed so far.

The main directive used to create rules is SecRule, whose syntax is as follows:

SecRule VARIABLES OPERATOR [ACTIONS]

- VARIABLES: Specify which places to check in an HTTP transaction. mod_security preprocesses raw transaction data, making it easy for rules to focus on the logic of detection. Currently, variables are divided into request, server, and response variables, parsing flags and time variables. You can use multiple variables in a single rule with the | operator.
- OPERATORS: Specify a regular expression, pattern or keyword to be checked in the variable(s). There are four types of operators: *string-*

matching, *numerical*, *validation* and *miscellaneous*operators. Operators always begin with a @ character, and are always followed by a space.

♣ ACTIONS: Specify what to do if the rule evaluates to “true” — step on to another rule, display an error message, or any other task. Actions are divided into seven
categories: *disruptive*, *flow*,*metadata*, *variable*, *logging*, *special* and *miscellaneo us* actions.

Here is a simple example of a rule:

```
SecRule ARGS|REQUEST_HEADERS "@rx <script" id:101,msg: 'XSS
Attack', severity:ERROR,deny,status:404
```

Here, ARGS and REQUEST_HEADERS are variables (request parameters and request headers, respectively); @rx is the operator used to match a pattern in the variables (here, this pattern is<script); id, msg, severity, deny and status are all actions to be performed if the pattern is matched. This rule is used to avoid XSS attacks by checking for a <script pattern in the request parameters and header, and generates an 'XSS Attack' message. The id:101 is given to the rule; it will deny any matching request with a 404 status response.

Let’s look at another example, for more clarity:

```
SecRule ARGS:username "@streq admin" chain,deny
SecRule REMOTE_ADDR "!@streq 192.168.1.1"
```

This is an example of chaining two rules, and the transfer of control to another rule if the first rule holds true. The first rule checks for the string admin in the request’s username parameter. If found, the second rule will be activated, which denies all such requests that are not from the192.168.1.1 IP address. Thus, chaining rules help to create complex rules.

Now, writing filtering rules for each attack will be very cumbersome, and also prone to human error. Here, mod_security provides users with another directive, SecFilter. This looks for a keyword in the request. It will be applied to the first line of the request (the one that looks like GET /index.php?parameter=value HTTP/1.0). In case of POST requests, the body of the request will be searched too (provided request body buffering is enabled). All pattern matches are case-insensitive, by default. The syntax for SecFilter is SecFilter KEYWORD.

Rules against major attacks

Let’s look at some rules to prevent major attacks on Web applications.

SQL injection

Suppose you have an application that is vulnerable to SQL-injection attacks. An attacker could try to delete all records from a MySQL table, like this:

```
http://www.example.com/login.php?user=arpit';DELETE%20FROM%20users--
```

This can be prevented with the following directive:

```
SecFilter "delete[[:space:]]+from"
```

Whenever such a request is caught by the filter, something similar to the following code is logged to audit_log:

```
========================================
Request: 192.168.0.207 - - [04/Jul/2006:23:43:00 +1200] "GET
/login.php?user=tom';DELETE%20FROM%20users-- HTTP/1.1" 500 1215
Handler: (null)
----------------------------------------
GET /login.php?user=arpit';DELETE%20FROM%20users-- HTTP/1.1
```

```
Host: 192.168.0.100
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.4)
Gecko/20060508 Firefox/1.5.0.4
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/pn
g,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
mod_security-message: Access denied with code 500. Pattern match
"delete[[:space:]]+from" at THE_REQUEST
mod_security-action: 500

HTTP/1.1 500 Internal Server Error
Last-Modified: Fri, 21 Oct 2005 14:30:18 GMT
ETag: "8238-4bf-833a5280"
Accept-Ranges: bytes
Content-Length: 1215
Connection: close
Content-Type: text/html
```

In response to the attack, SecFilterDefaultAction is applied (the request is denied, logged, and the attacker gets a 500 error). If you want a different action to take place (like, redirect the request to a HTML page that can provide customised warning content), you can specify this in the rule, as follows:

```
SecFilter "delete[[:space:]]+from"
log,redirect:http://example.com/invalid_request.html
```

To prevent more SQL injection attacks, we can add a few other directives like:

```
SecFilter "insert[[:space:]]+into"
SecFilter "select.+from"
SecFilter "drop[[:space:]]table"
SecFilter create[[::space:]]+table
SecFilter update.+set.+=
SecFilter union.+select
SecFilter or.+1[[:space:]]*= [[:space:]]1
SecFilter '.+--
SecFilter xp_enumdsn
SecFilter xp_cmdshell
SecFilter xp_regread
SecFilter xp_regwrite
SecFilter xp_regdeletekey
```

The last five are particularly used for MS SQL server-specific injection attacks.

The only problem with SecFilter is that it scans the whole request instead of particular fields. Here, SecFilterSelective is useful; it allows you to choose exactly what to search. The syntax is:

```
SecFilterSelective LOCATION KEYWORD [ACTIONS]
```

Here, LOCATION decides which area of the request to be filtered. Hence, for SQL injection, you can also use:

```
SecFilterSelective SCRIPT_FILENAME "login.php" chain
SecFilterSelective ARG_user "!^[a-zA-Z0-9\.@!]{1,10}$"
```

The above code will validate the user parameter, and allow only the white-list of characters we have given. If for some reason you cannot take this approach, and must use a *deny-what-is-bad*method, then at least remove single quotes ('), semicolons (;), dashes, hyphens (-), and parenthesis (()).

XSS attacks

For XSS attacks, we can use the following directives:

```
SecFilter "<(.|\n)+>"
SecFilter "<[[:space:]]*script"
SecFilter "<script"
SecFilter "<.+>"
```

And also, some additional filters like:

```
SecFilterSelective THE_REQUEST "<[^>]*meta*\"?[^>]*>"
SecFilterSelective THE_REQUEST "<[^>]*style*\"?[^>]*>"
SecFilterSelective THE_REQUEST "<[^>]*script*\"?[^>]*>"
SecFilterSelective THE_REQUEST "<[^>]*iframe*\"?[^>]*>"
SecFilterSelective THE_REQUEST "<[^>]*object*\"?[^>]*>"
SecFilterSelective THE_REQUEST "<[^>]*img*\"?[^>]*>"
SecFilterSelective THE_REQUEST "<[^>]*applet*\"?[^>]*>"
SecFilterSelective THE_REQUEST "<[^>]*form*\"?[^>]*>"
```

Though these filters will detect a large number of XSS attacks, they are not foolproof. Due to the multitude of different scripting languages, it is possible for an attacker to create many different methods for implementing an XSS attack that would bypass these filters. Hence, here it is advised that you also keep on adding your own filters.

To protect against an XSS attack done via PHP session cookies, you can use the following:

```
SecFilterSelective ARG_PHPSESSID "!^[0-9a-z]*$"
SecFilterSelective COOKIE_PHPSESSID "!^[0-9a-z]*$"
```

Command execution attacks

For command execution attacks, you can use the following directives:

```
SecFilter /etc/password
SecFilter /bin/ls
```

Here, the attacker may try to use a string like `/bin/./sh` to bypass the filter — but `mod_security`automatically reduces `/./` to `/` and `//` to `/`, and also decodes URL-encoded characters. You can also use the white-list approach:

```
SecFilterSelective SCRIPT_FILENAME "directory.php" chain
SecFilterSelective ARG_dir "!^[a-zA-Z/_-\.0-9]+$"
```

This chained rule-set will only allow letters, numbers, underscore, dash, forward slash, and period in the `dir` parameter. Filtering out command directory names is also a good option, and can be done as follows:

```
SecFilterSelective THE_REQUEST "/^(etc|bin|sbin|tmp|var|opt|dev|kernel)$/"
SecFilterSelective ARGS "bin/"
```

Session fixation

During session fixation, in one of its phases, the attacker needs to somehow inject the desired session ID into the victim’s browser. We can mitigate these issues by implementing the following:

```
# Weaker XSS protection, but allows common HTML tags
SecFilter "<[[:space:]]*script"
```

```
# Prevent XSS attacks (HTML/Javascript injection)
SecFilter "<.+>"
# Block passing Cookie/Session IDs in the URL
SecFilterSelective THE_REQUEST "(document\.cookie|Set-
Cookie|SessionID=)"
```

Directory traversal attacks

For path/directory traversal attacks, the following directives are mostly used:

```
SecFilter "\.\./"
SecFilterSelective SCRIPT_FILENAME "/scripts/foo.cgi" chain
SecFilterSelective ARG_home "!^[a-zA-Z].{15,}\.txt"
```

The last two filters are chained, and will reject all parameters to the home argument that is a filename of more than 15 alpha characters, and that doesn't have a .txt extension.

Similarly, you can prevent predictable resource location attacks also, and protect against sensitive file misuse, with two recommended solutions. First, remove files that are not intended for public viewing from all Web server-accessible directories. After this, you can create security filters to identify if someone probes for these files:

```
SecFilterSelective REQUEST_URI "^/(scripts|cgi-local|htbin|cgibin|cgis|win-
cgi|cgi-win|bin)/"
SecFilterSelective REQUEST_URI ".*\.(bak|old|orig|backup|c)$"
```

These two filters will deny access to both — unused, but commonly scanned for directories, and files with common backup extensions.

Web pages that are dynamically created by the directory-indexing function will have a title that starts with “Index of /”. We can use this as a signature, and add the following directives to catch and deny access to this data:

```
SecFilterScanOutput On
SecFilterSelective OUTPUT "\<title\>Index of /"
```

Information leakage

Here, we are introduced to the OUTPUT filtering capabilities of mod_security, which you should enable by adding SecFilterScanOutput On in the configuration file. We can easily set up a filter to watch for common database error messages being sent to the client, and then generate a generic 500 status code instead of the verbose message:

```
SecFilterScanOutput On
SecFilterSelective OUTPUT "An Error Has Occurred" status:500
SecFilterSelective OUTPUT "Fatal error:"
```

Output filtering can also be used to detect successful intrusions. These rules will monitor output, and detect typical keywords resulting from a command execution on the server.

```
SecFilterSelective OUTPUT "Volume Serial Number"
SecFilterSelective OUTPUT "Command completed"
SecFilterSelective OUTPUT "Bad command or filename"
SecFilterSelective OUTPUT "file(s) copied"
SecFilterSelective OUTPUT "Index of /cgi-bin/"
SecFilterSelective OUTPUT ".*uid\=\("
```

Secure file uploads

mod_security is capable of intercepting files uploaded through POST requests and multi-part/form-data encoding through PUT requests. It will always upload files to a temporary directory. You can choose the directory using the SecUploadDir directive:

```
SecUploadDir /tmp
```

It is better to choose a private directory for file storage, somewhere that only the Web server user account is allowed access. Otherwise, other server users may be able to access files uploaded through the Web server. You can choose to execute an external script to verify a file before it is allowed to go through to the application. The SecUploadApproveScript directive enables this, like the following example:

```
SecUploadApproveScript /usr/local/apache/bin/upload_verify.pl
```

RFI attacks

RFI attacks are generally easy to detect, with something like the following directive:

```
SecRule ARGS “@rx (?i)^(f|ht)tps?://([^/])” msg:’Remote File Inclusion attack’
# To detect inclusions containing IP address
SecRule ARGS "@rx (ht|f)tps?://([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])" msg:'Remote File Inclusion attack'
#To detect inclusions containing PHP function ‘include()’
SecRule ARGS "@rx \binclude\s*\([\w|\s]*(ht|f)tps?://" "msg:'Remote File Inclusion'"
# To detect inclusion ending with ‘?’
SecRule ARGS "@rx (ft|htt)ps?.*\?+$" msg:'Remote File Inclusion'
```

Miscellaneous security features

You can also block IP addresses by the following command:

```
SecFilterSelective "REMOTE_ADDR" "^192.168.1.1$"
```

If you have an input field URL in your comment form, and you want to scan the value of URL for the string c99, you do it as follows:

```
SecFilterSelective "ARG_url" "c99"
```

The following configuration helps fight HTTP fingerprinting, and accepts only valid protocol versions:

```
SecFilterSelective SERVER_PROTOCOL !^HTTP/(0\.9|1\.0|1\.1)$
```

The following configuration allows supported request methods only, and helps fight XST attacks:

```
SecFilterSelective REQUEST_METHOD !^(GET|HEAD|POST)$
```

Often during the reconnaissance phase, attackers look for the Web server identity and version. Web servers typically send their identity with every HTTP response, in the Server header. Apache is particularly helpful here; it not only sends its name and full version, by default, but also allows server modules to append their versions. Here, you can confuse the attackers by using something like:

```
SecServerSignature "Microsoft-IIS/5.0"
```

PHP code cannot be injected directly, but it may be possible to have code recorded on disk to be executed later, using an LFI attack. The following rule will detect such an injection attempt, but will ignore XML documents, which use similar syntax:

```
SecRule ARGS "@rx <\?(?!xml)"
```

Logging

There are three places where, depending on the configuration, you may find mod_securitylogging information:

♣ mod_security debug log: If enabled via the SecFilterDebugLevel and SecFilterDebugLogdirectives, it contains a large number of entries for every request processed. Each log entry is associated with a log level, which is a number from 0 (no messages at all) to 4 (maximum logging). You normally keep the debug log level at 0, and increase it only when you are debugging your rule set.

♣ Apache error log: Some of the messages from the debug log will make it into the Apache error log (even if you set mod_security debug log level to 0). These are the messages that require an administrator’s attention, such as information about requests being rejected.

♣ mod_security audit log: When audit logging is enabled (using the SecAuditEngine andSecAuditLog directives), mod_security can record each request (and its body, provided request body buffering is enabled) and the corresponding response headers.

Here is an example of an error message resulting from invalid content discovered in a cookie:

```
[Tue Jun 26 17:44:36 2011] [error] [client 127.0.0.1]
mod_security: Access denied with code 500. Pattern match "!(^$|^[a-zA-Z0-9]+$)"
at COOKIES_VALUES(sessionid) [hostname "127.0.0.1"]
[uri "/test.php"] [unique_id 3434fvnij54jktynv45fC8QQQQAB]
```

The message indicates that the request was rejected (“Access denied”) with an HTTP 500response because the content of the cookie sessionid contained content that matched the pattern !(^$|^[a-zA-Z0-9]+$). (The pattern allows a cookie to be empty, but if it is not, it must consist only of one or more letters and digits.)

Note: I once again stress that neither LFY nor myself are responsible for the misuse of the information given here. Any attack techniques described here are meant to give you the knowledge that you need to protect your own infrastructure. Please use the tools and techniques sensibly.

This article has just scratched the surface of mod_security. For more details on rule writing and other important directives, please refer to *ModSecurity Handbook* by Ivan Ristic — a must-read book for anyone interested in this topic.

We will deal with other ways to secure Apache

Source : http://www.opensourceforu.com/2011/08/securing-apache-part-10-mod_security/