mod_security

http://www.modsecurity.org

Reference Manual v1.8.4

29 July 2004

Table of Contents

Reference Manual v1.8.4 1
- Introduction 5
- Licensing 5
- Acknowledgments 5
- Contact 5
- Installation 6
  - CVS Access 6
  - Download 6
    - Nightly snapshot 6
    - Releases 7
  - Installing from source 7
    - DSO 7
    - Static installation with Apache 1.x 7
  - Installing from binary 7
    - Apache 1.x 8
    - Apache 2.x 8
- Configuration 8
  - Turning filtering on and off 8
  - POST scanning 8
    - Turning buffering off dynamically 9
  - Chunked transfer encoding 9
  - Default action 9
  - Filter inheritance 10
  - URL Encoding validation 10
  - Unicode encoding validation 10
  - Byte range check 11
  - Allowing others to see mod_security 11
- Debugging 12
- Request filtering 12
  - Simple filtering 13
  - Path normalization 13
    - Null byte attack prevention 13
  - Regular expressions 14
    - Inverted expressions 14

Advanced filtering..... 14
Argument filtering exceptions..... 16
Cookies..... 16
Output filtering..... 17
Actions..... 18
Specifying actions..... 18
Built-in actions..... 19
pass..... 19
allow..... 19
deny..... 19
status..... 19
redirect..... 19
exec..... 19
log..... 20
nolog..... 20
skipnext..... 20
chain..... 20
pause..... 20
Request headers added by mod_security..... 21
Handling rule matches using ErrorDocument..... 21
Making mod_security talk to your firewall..... 21
File upload support..... 22
Choosing where to upload files..... 22
Verifying files..... 22
Storing uploaded files..... 22
Upload memory limit..... 22
Other features..... 23
Server identity masking..... 23
Chroot support..... 23
Standard approach..... 23
The mod_security way..... 24
Required module ordering for chroot support (Apache 1.x)..... 25
Required module ordering for chroot support (Apache 2.x)..... 25
How mod_security chroot works..... 26
Solving common security problems..... 26
Directory traversal..... 26
Cross site scripting attacks..... 27

SQL/database attacks..... 27
Operating system command execution..... 27
Buffer overflow attacks..... 28
Custom logging..... 28
Audit logging..... 28
Unique request identifiers..... 29
Choosing what to log..... 29
The testing utility..... 30
Technology specific notes..... 31
PHP..... 31
Preventing register_global problems..... 31
Additional Examples..... 32
Parameter checking..... 32
File upload..... 32
Securing FormMail..... 32
Performance..... 32
Speed..... 33
Memory consumption..... 33
Other things to watch for..... 33
Known issues..... 33
Other resources..... 33
Appendix A: Recommended Configuration..... 34

Introduction

ModSecurity is an open source intrusion detection and prevention engine for web applications. It operates embedded into the web server, acting as a powerful umbrella - shielding applications from attacks.

ModSecurity integrates with the web server, increasing your power to deal with web attacks. Some of its features worth mentioning are:

- **Request filtering**; incoming requests are analysed as they come in, and before they get handled by the web server or other modules.
- **Anti-evasion techniques**; paths and parameters are normalised before analysis takes place in order to fight evasion techniques.
- **Understanding of the HTTP protocol**; since the engine understands HTTP, it performs very specific and fine granulated filtering.
- **POST payload analysis**; the engine will intercept the contents transmitted using the POST method, too.
- **Audit logging**; full details of every request (including POST) can be logged for later analysis.
- **HTTPS filtering**; since the engine is embedded in the web server, it gets access to request data after decryption takes place.

Licensing

Acknowledgments

This module would not be possible without the fine people who have created the Apache Web server, and the fine people who have spent many hours building Apache modules I used to learn Apache module programming from. My special thanks goes to the authors of the mod_rewrite from whom I've learned the most (thanks Ralf).

Contact

ModSecurity is being developed by Ivan Ristic. Comments and questions are welcome. Please send your emails to ivanr@webkreator.com.

Installation

Before you begin with installation you will need to choose your preferred installation method. First you need to choose whether to install the latest mod_security version directly from CVS (latest features, but possibly unstable) or use the latest stable release (stable but with less features).

If you choose a stable release, it might be possible to install mod_security from binary, or compile it from the source code.

Finally, when compiling from source you need to choose a statically compiled module that integrates with the web server or a shared library module that is loaded into the web server at run time.

The next few pages will give you more information on benefits of choosing one way over another.

CVS Access

If you want to access the up to date version of the module you need to get it through CVS. The list of changes made from the last release is normally available on the web site. CVS for mod_security is being hosted by SourceForge, http://www.sf.net. You can access it directly or view if through web using this address: http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/mod-security/

To download the source code to your computer you need to execute the following two commands:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/mod-security login
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/mod-security co
mod_security
```

The first line will log you in as an anonymous user, and the second will download all files available in the mod_security repository.

Download

Nightly snapshot

If you don't like CVS but you still want the latest version you can download the latest nightly tarball from the following address:

```
http://www.modsecurity.org/download/snapshot/mod_security-snapshot.tar.gz
```

New features are added to mod_security one by one, with regression tests being run after each change. This should ensure that the version available from CVS is always usable.

Releases

You can download stable mod_security releases from the web site, go to http://www.modsecurity.org/download/. You may even be able to find binary releases for some platforms.

Installing from source

When installing from source you have two choices: to install the module into the web server itself, or to compile mod_security into a dynamic shared object (DSO).

DSO

Installing as DSO is easier, and the procedure is the same for both Apache branches:

1. Unpack mod_security source code

2. Compile the module into the web server:

```
/apachehome/bin/apxs -cia mod_security.c
```

3. Stop and start the server

```
/apachehome/bin/apachectl stop
/apachehome/bin/apachectl start
```

Note: I've had reports from people using platforms that do not have apxs installed. If you have access to such a platform please read the documentation from your vendor and let me know how are Apache modules added (on some RedHat platforms you need to install the package http-devel to get access to apxs).

Static installation with Apache 1.x

When a module is compiled statically, it gets embedded into the body of the web server. This method results in a slightly faster executable but the the compilation method is a bit more complicated, too:

1. Copy the file mod_security.c to /src/modules/extra

2. Configure Apache distribution with two additional configuration options:

```
--activate-module=src/modules/extra/mod_security
--enable-module=security
```

3. Compile and install as usual

Installing from binary

In some circumstances, you will want to install the module as a binary. At the moment I only make Windows binaries available for download. This may change as mod_security releases become further apart.

Apache 1.x

1. Copy the mod_security.so (on Unix) or the mod_security.dll (on Windows) file to libexec/ and

2. Add the following line to the httpd.conf file

```
LoadModule security_module    libexec/mod_security.so
```

Apache 2.x

1. Copy the mod_security.so (on Unix) or the mod_security.dll (on Windows) file to modules/ and

2. Add the following line to the httpd.conf file (again, use mod_security.dll on Windows).

```
LoadModule security_module    modules/mod_security.so
```

Configuration

Mod_security configuration directives are added to your configuration file directly. Normally, this is the file httpd.conf.

Turning filtering on and off

Filtering engine is turned off by default. To use it, you need to turn it on:

```
SecFilterEngine On
```

Supported parameter values for this parameter are:

- On – analyse every request
- Off – do nothing
- DynamicOnly – analyse only requests generated dynamically at runtime. Using this option will prevent your web server from using precious CPU cycles on checking access to static files. To understand how mod_security decides what is a dynamically generated request please read the section "Choosing what to log" (the same principle applies there).

POST scanning

POST payload scanning is turned off by default. To use it, you need to turn it on:

```
SecFilterScanPOST On
```

ModSecurity supports two encoding types for the request body:

- **application/x-www-form-urlencoded** - this is an encoding used to transfer data from forms

- **multipart/form-data** - (since 1.8dev1) encoding used for file upload

To make sure that only requests with these two encoding types are accepted by the web server, add the following line to your configuration file:

```
SecFilterSelective HTTP_Content-Type "!^(|application/x-www-form-
urlencoded|multipart/form-data)$"
```

Turning buffering off dynamically

It is possible to turn POST scanning off on per-request basis. If mod_security sees that an environment variable MODSEC_NOPOSTBUFFERING is defined it will not perform POST buffering. For example, to turn POST buffering off for file uploads use the following:

```
SetEnvIfNoCase Content-Type "^multipart/form-data"
"MODSEC_NOPOSTBUFFERING=Do not buffer file uploads"
```

The value of the MODSEC_NOPOSTBUFFERING variable will be written to the debug log, so you can put in there something that will tell you why was buffering turned off.

Chunked transfer encoding

The HTTP protocol supports a method of request transfer where the size of the payload is not known in advance. The body of the request is delivered in chunks (hence the name). As far as I am aware no browser is using the chunked encoding to send requests. ModSecurity does not support chunked requests at this time; when a POST request is made with chunked encoding it will ignore the body of the request. **Add the following line to your configuration to prevent attackers to exploit this weakness**:

```
SecFilterSelective HTTP_Transfer-Encoding "!^$"
```

Note: This will not affect your ability to send responses using chunked transfer encoding.

Default action

Whenever a filter is matched against a request, an action (or a series of actions) is taken. Individual filters can each have their own actions but in practice you will want to define a set of actions for all filters. You can do this with the configuration directive SecFilterDefatultAction. For example:

```
SecFilterDefaultAction "deny,log,status:404"
```

This directive accepts only one parameter, a series of actions separated with commas. Actions you specify here will be applied on every filter match, where individual actions were not defined.

Filter inheritance

Filters defined in parent folders are usually inherited by nested Apache configurations. This is behavior is acceptable (and required) in most cases, but not all. Sometimes you need to relax checks in some part of the site. By using the following directive:

```
SecFilterInheritance Off
```

you can instruct mod_security to disregard parent filters so that you can start from the scratch.

Note: This is the only option which will not be inherited by a nested Apache configuration.

URL Encoding validation

Special characters need to be encoded before they can be transmitted in the URL. Any character can be replaced using the three character combination %XY, where XY is an hexadecimal character code (see http://www.rfc-editor.org/rfc/rfc1738.txt for more details). Hexadecimal numbers only allow letters A to F, but attackers sometimes use other letters in order to trick the decoding algorithm. Mod_security checks all supplied encodings in order to verify they are valid.

You can turn URL encoding validation with the following line:

```
SecFilterCheckURLEncoding On
```

Note: This directive will not check encoding in a POST payload when "multipart/form-data" encoding (file upload) is used.

Unicode encoding validation

Unicode encoding validation first appeared in version 1.6. Normally, it is disabled by default. You should turn it on if your application or the underlying operating system accept/understand Unicode.

Note: More information on Unicode and UTF-8 encoding can be found in RFC 2279 (http://www.ietf.org/rfc/rfc2279.txt).

```
SecFilterCheckUnicodeEncoding On
```

This feature will assume UTF-8 encoding and check for three types of errors:

- **Not enough bytes.** UTF-8 supports two, three, four, five, and six byte encodings. ModSecurity will locate cases when a byte or more is missing.
- **Invalid encoding.** The two most significant bits in most characters are supposed to be fixed to 0x80. Attackers can use this to subvert Unicode decoders.

- **Overlong characters.** ASCII characters are mapped directly into the Unicode space and are thus represented with a single byte. However, most ASCII characters can also be encoded with two, three, four, five, and six characters thus tricking the decoder into thinking that the character is something else (and, presumably, avoiding the security check).

Byte range check

You can force requests to consist only of bytes from a certain byte range. This can be useful to avoid stack overflow attacks (since they usually contain "random" binary content).

To only allow bytes from 32 to 126 (inclusive), use the following directive:

```
SecFilterForceByteRange 32 126
```

Default range values are 0 and 255, i.e. all byte values are allowed.

Note: This directive will not check byte range in a POST payload when "multipart/form-data" encoding (file upload) is used.

Allowing others to see mod_security

Normally, attackers won't be able to tell whether your web server is running mod_security or not. You can give yourself away by sending specific messages, or by using unusual HTTP codes (e.g. 406). If you want to stay hidden your best bet is to use HTTP 500, which stands for "Internal Server Error". Attackers that encounter such a response might think that your application has crashed.

There is another school of thought on this matter, which says that you should not hide the fact that you are running mod_security. The theory says that if they see it they will know you pay attention and that breaking into will be very difficult. And that they will go away looking for a weaker target. Or maybe they will become more determined and challenged.

By default Apache will return the information on itself with every request it serves. Mod_security keeps quiet buy default, but you can allow others to see it by adding the following line to your configuration:

```
SecServerResponseToken On
```

The result will be similar to this:

```
[ivanr@wkx conf]$ telnet 0 8080
Trying 0.0.0.0...
Connected to 0.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.1 406 Not Acceptable
Date: Mon, 19 May 2003 18:13:51 GMT
```

```
Server: Apache/2.0.45 (Unix) mod_security/1.5
Content-Length: 351
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>406 Not Acceptable</title>
</head><body>
<h1>Not Acceptable</h1>
<p>An appropriate representation of the requested resource / could not be
found on this server.</p>
<hr />
<address>Apache/2.0.45 (Unix) mod_security/1.4.2 Server at wkx.dyndns.org
Port 80</address>
</body></html>

Connection closed by foreign host.
```

You should note that Apache itself supports two runtime directives, ServerTokens and ServerSignature. Using these directives you can completely hide the information on your server, no matter what you told mod_security to do.

Debugging

Use SecFilterDebugLog to choose a file where debug output will be written. If the parameter does not start with the forward slash, Apache home path will be pre-pended to it.

```
SecFilterDebugLog logs/modsec_debug_log
```

You can control how detailed the debug log is with SecFilterDebugLevel:

```
SecFilterDebugLevel 4
```

- 0 - none
- 1 - significant events (these will also be reported in the error_log)
- 2 - info messages
- 3 - more detailed info messages

Note: Yes, there are several other logging levels but, believe me, you don't want to use them :) They are only there for me, to aid me in development.

Request filtering

When mod_security is On, every incoming request is intercepted and analysed prior to fulfillment. The analysis first performs a series of built-in checks in order to validate the request. These checks can be controlled using configuration directives, as you

have already seen. The other part of the analysis consists from a series of user-defined filters that are matched against the request. Whenever there is a positive match, certain actions are taken.

Simple filtering

The most simplest form of filtering mod_security offers is, well, simple. It looks like this:

```
SecFilter KEYWORD
```

For each simple filter like this, mod_security will look for the keyword in the request. The search is pretty broad; it will be applied to the first line of the request (the one that looks like this "GET /index.php?parameter=value HTTP/1.0"). In case of POST requests, the body will be searched too (provided you turn the SecFilterScanPOST On).

Path normalization

Filter are not applied to raw request data, but on a normalized copy. We do this because attackers can (and do) apply various evasion techniques to avoid detection. For example, you might want to setup a filter that detects shell command execution:

```
SecFilter /bin/sh
```

But the attacker may use /bin/./sh and avoid the filter.

These transformations are applied in the current version of mod_security:

- On Windows only, convert \ to /
- Reduce /./ to /
- Verify URL encoding (optional)
- Only allow a byte range (optional)
- Decodes URL encoding
- Reduce // to /

Null byte attack prevention

Null byte attacks try to confuse C/C++ based software and trick it into thinking that a string ends sooner than it actually does. This type of an attack is rejected with the SecFilterByteRange filter. Problems can arise if you are not using this filter, and mod_security is vulnerable to this type of attacks.

To fight this, mod_security looks for null bytes during the decoding phase and converts them into spaces. So, where before this filter:

```
SecFilter hidden
```

would not detect the word hidden in this request (* with SecFilterByteRange turned off):

```
GET /one/two/three?p=visible%00hidden
```

with mod_security 1.7 and above it works as you would have expected.

Regular expressions

Remember the simple filtering facility, the one that looked like this:

```
SecFilter KEYWORD [ACTIONS]
```

There is a twist to it. When specifying the keyword you are not limited to a single word. Instead you can put any regular expression that you want. To make most out of this (now) powerful tool you need to understand regular expressions well. I recommend that you start with one of the following resources:

- Perl-compatible regular expressions man page, http://www.pcre.org/pcre.txt
- Perl Regular Expressions, http://www.perldoc.com/perl5.6/pod/perlre.html
- Mastering Regular Expressions, http://www.oreilly.com/catalog/regex/
- Google search on regular expressions, http://www.google.com/search?q=regular%20expressions

Note: Be careful with regular expressions of type:

```
(A|B)+
```

Applying such expressions over large quantities of text can lead to stack overflow on platforms that have small stack size (e.g. Windows). Stack overflow will result in non-exploitable segfault.

Inverted expressions

If exclamation mark is the first character of a regular expression, the filter will treat that regular expression as inverted. For example, the following:

```
SecFilter "!php"
```

will reject all requests that do not contain the word "php".

Advanced filtering

While SecFilter allows you to start quickly, you will soon discover that the search it performs is too broad, and doesn't work very well. Another directive,

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

allows you to choose exactly where you want the search to be performed. The KEYWORD and the ACTIONS bits are the same as in SecFilter. The LOCATION bit

requires further explanation.

The LOCATION parameter consist of a series of location identifiers separated with a pipe. For example:

```
SecFilterSelective "REMOTE_ADDR|REMOTE_HOST" KEYWORD
```

will search only the IP address of the client and the host name. The list of possible location includes all CGI variables, and some more. Here is the full list:

- REMOTE_ADDR
- REMOTE_HOST
- REMOTE_USER
- REMOTE_IDENT
- REQUEST_METHOD
- SCRIPT_FILENAME
- PATH_INFO
- QUERY_STRING
- AUTH_TYPE
- DOCUMENT_ROOT
- SERVER_ADMIN
- SERVER_NAME
- SERVER_ADDR
- SERVER_PORT
- SERVER_PROTOCOL
- SERVER_SOFTWARE
- TIME_YEAR
- TIME_MON
- TIME_DAY
- TIME_HOUR
- TIME_MIN
- TIME_SEC

- TIME_WDAY
- TIME
- API_VERSION
- THE_REQUEST
- REQUEST_URI
- REQUEST_FILENAME
- IS_SUBREQ

There are some special locations:

- POST_PAYLOAD – filter the body of the POST request
- ARGS - filter arguments, the same as "QUERY_STRING|POST_PAYLOAD"
- ARGS_NAMES – variable/parameter names only
- ARGS_VALUES – variable/parameter values only
- COOKIES_NAMES - cookie names only
- COOKIES_VALUES - cookie values only

And even more special:

- HTTP_header – search request header "header"
- ENV_variable – search environment variable "variable"
- ARG_variable – search request variable/parameter "variable"
- COOKIE_name - search cookie with name "name"

Argument filtering exceptions

The ARG_variable location names support inverted usage when used together with the ARG location. For example:

```
SecFilterSelective "ARGS|!ARG_param" KEYWORD
```

will search all arguments except the one named "param".

Cookies

Mod_security supports cookies fully. There are two directives that control how we behave while parsing cookies.

By default, mod_security will not validate cookie format. You can turn format

validation On by using the SecFilterCheckCookieFormat directive:

```
SecFilterCheckCookieFormat On
```

By default, mod_security will try to normalize cookie names and values. This is because many platforms encode them before putting them into a cookie. However, this behavior is not appropriate in all cases. What often happens is that some cookies, while not being encoded, contain a plus ('+') character. It gets decoded to a space, and that can lead to a triggered rule. To turn normalization off use the SecFilterNormalizeCookies directive.

```
SecFilterNormalizeCookies Off
```

Output filtering

Starting with 1.7, the Apache2 version of mod_security supports output filtering. There is a way to add this feature to Apache 1.x too but it requires a lot of work; therefore it will be added later if there is a need for it. In the meantime, consider using Apache 2.x if you really need this feature.

First you need to turn the output filtering on:

```
SecFilterScanOutput On
```

After that, simply add selective filters using the variable OUTPUT:

```
SecFilterSelective OUTPUT "credit card numbers"
```

Those who have perhaps followed my columns at http://www.webkreator.com/php/ know that I am somewhat obsessed with the inability of PHP to prevent fatal errors. I have gone to great lengths to prevent fatal errors from spilling to end users (see http://www.webkreator.com/php/configuration/handling-fatal-and-parse-errors.html) but now, finally, I don't have to worry any more about that.

```
SecFilterSelective OUTPUT "Fatal error:" deny,status:500
ErrorDocument 500 /php-fatal-error.html
```

You should note that although you can put output filters wherever you want, they are **not** executed at the same time with other filters. All other filters are executed before the request is allowed through, but output scanning is performed only after the request has completed.

Note: At the moment, skipnext, and chained actions do not work with output filters.

As of 1.7.2 mod_security offers a SecFilterOutputMimeTypes directive to allow output filtering optimizations based on the mime type of the output. Using this directive you can restrict output filtering only to those types of files where filtering make sense.

```
SecFilterOutputMimeTypes "(null) text/html text/plain"
```

Configured as in example above mod_security will apply output filters to plain text files, html files, and files where the mime type is not specified "(null)".

Note: Using output buffering will make mod_security keep the whole of the page output in memory, no matter how large it is. The memory consumption is **over twice the size** of the page. In a future version I will introduce a configuration option to limit the maximum size of the output.

Actions

There are several types of actions:

- A primary action will make a decision whether to continue with the request or not. There can exist only one primary action. If you put several primary actions in the parameter, the last action to be seen will be executed. Primary actions are deny, pass, and redirect..
- Secondary actions will be performed on a filter match independently on the decision made by primary actions. There can be any number of secondary actions. One secondary action is exec.
- Flow actions can change the flow of rules, causing mod_security to jump to another rule, or to skip one or several rules.
- Parameters are not really actions, but a method of attaching parameters to filters. Some of this parameters can be used by real actions. For example status supplies the response code to the primary action deny.

Specifying actions

There are three places where you can put actions. One is the SecFilterDefaultAction directive, where you define actions you want executed on every filter match:

```
SecFilterDefaultAction "deny,log,status:500"
```

This example defines three examples separated using commas. First two actions consist of a single word. Third action requires a parameter. Use double colon to separate the parameter from the action name.

You can also specify per-filter actions. Both filtering directives accept a set of actions as an optional parameter. Examples of why you would want this can be found further in the text.

Note: In 1.6 and below, when per-filter action was specified it was "empty" by default, i.e. on a match it would not deny the request. Many people did not expect this. Because allowing a request can be more harm than blocking it I decided to slightly alter the behavior here. With 1.7 and above if you don't explicitly allow the request - it will be rejected.

Built-in actions

pass

Allow request to continue on filter match. This action is useful when you want to log a filter match but otherwise do not want to take action.

```
SecFilter KEYWORD "pass,log"
```

allow

This is a stronger version of the previous filter. After this action is performed the request will be allowed through and no other filters will be tried:

```
# let those who are in the know through, no questions asked
SecFilterSelective HTTP_USER_AGENT "The meaning of life is 42" allow
```

deny

Interrupt request processing when a filter is matched. Unless status action is used, mod_security will immediately return a HTTP 500 error code. If a request is denied a header "mod_security-action" will be added to the list of request headers. It will contain the status code sent by mod_security.

status

Use the supplied HTTP status code when request is denied:

```
SecFilter KEYWORD "status:404"
```

will return a "Page not found" if triggered. Apache ErrorDocument directive works well here. If you have previously defined a custom error page for a given status then it will be displayed to the user.

redirect

Redirect the user to the given URL. For example:

```
SecFilter KEYWORD "redirect:http://www.modsecurity.org"
```

This configuration directive will always override HTTP status code, or the deny keyword.

exec

Execute a binary on filter match. Full path to the binary is required:

```
SecFilter KEYWORD "exec:/home/ivanr/report-attack.pl"
```

This directive does not effect the primary action (allow or deny). This action will always call script with no parameters, but providing all information in the environment. All the usual CGI environment variables will be there.

You can have one binary executed per filter match. Execution will add a header "mod_security-executed" to the list of request headers.

Note: Forking a threaded process will result in threads being replicated in a new process. Forking can therefore incur larger overhead in multithreaded operation.

log

Log filter match to Apache error log.

nolog

Do not log filter match to Apache error log.

skipnext

This action allows you to skip over one or more rules. You will use this action when you establish that there is no need to perform some tests on a particular request. By default, the action will skip over the next rule. It can jump any number of rules provided you supply the optional parameter:

```
SecFilterSelective ARG_p value1 skipnext:2
SecFilterSelective ARG_p value2
SecFilterSelective ARG_p value3
```

chain

Rule chaining allows you to chain several rules into a bigger test. Only the last rule in the chain will affect the request but in order to reach it, all rules before it must be matched too. Here is an example of how you might use this feature.

I wanted to restrict the administration account to log in only from a certain IP address. However, the administration login panel was shared with other users and I couldn't use the standard Apache features for this. So I used these two rules:

```
SecFilterSelective ARG_username admin chain
SecFilterSelective REMOTE_ADDR "!^YOUR_IP_ADDRESS_HERE$"
```

The first rule matches only if there exists a parameter username and its value is admin. Only then will the second rule be executed and it will try to match the remote address of the request to the single IP address. If there is no match (note the exclamation mark at the beginning) the request is rejected.

pause

Pause for the specified amount of milliseconds before responding to a request. This is useful to slow down or completely confuse some web scanners. Some scanners will give up if the pause is too long. This feature appeared first in 1.6.

Note: Be careful with this option as it comes at a cost. Every web server installation is configured with a limit, the maximal number of requests that may be served at at any

given time. Using a long delay time with this option may create a "voluntary" denial of service attack if the vulnerability scanner is executing requests in parallel (therefore many .

Request headers added by mod_security

Wherever possible, mod_security will add information to request headers, allowing your scripts to find and use them. Obviously, you will have to configure mod_security not to reject requests in order for your scripts to be executed at all. At a first glance it may be strange that I'm using the request headers (input headers) for this purpose instead of, for example, environment variables. Although environment variables would be more elegant, input headers are always visible to scripts executed using an ErrorDocument directive (see below) while environment variables are not.

This is the list of headers added:

- **mod_security-executed**; with the path to the binary executed
- **mod_security-action**; with the status code returned
- **mod_security-message**; the message about the problem detected, the same as the message added to the error log

Handling rule matches using ErrorDocument

If your mod_security configuration returns a HTTP status code 500, and you configure the Apache to execute custom scripts whenever this code ocurrs (e.g. ErrorDocument 500 /error500.php) you will be able to use your favourite scripting engine to respond to errors. The information on the error will be in the environment variables REDIRECT_* and HTTP_MOD_SECURITY_*.

Making mod_security talk to your firewall

In some cases, after detecting a particularly dangerous attack or a series of attacks you will want to prevent further attacks coming from the same source. You can do this by modifying the firewall to reject all traffic coming from a particular IP address.

This method can be very dangerous since it can result in a denial of service (DOS) attack. For example, an attacker can use a proxy to launch attacks. Rejecting all access from a proxy server can be very dangerous since all legitimate users will be rejected, too.

Since most proxies send information describing the original client (some information on this is available here http://www.webkreator.com/cms/view.php/1685.html, under the "Stop hijacking" header), we can try to be smart and find the real IP address. While this can work, consider the following scenario:

- The attacker is accessing the application directly but is pretending to be a proxy server, citing a random (or valid) IP address as the real source IP address. If we start rejecting requests based on that deducted information,

the attacker will simply change the IP address and continue. As a result we might have banned legitimate users while the attacker is still free searching for application holes.

Therefore this method can be useful only if you do not allow access to the application through proxies, or allow access only through proxies that are well known and, more importantly, trusted.

If you still want to ban requests based on IP address (in spite of all our warnings), you will need to write a small script that will executed on a filter match. The script should extract the IP address of the attacker from environment variables, and then make a call to iptables or ipchains to ban the IP address. We will include a sample script doing this with a future version of mod_security.

File upload support

Starting with 1.8dev1 mod_security fully supports the multipart/form-data support.

Choosing where to upload files

ModSecurity will always upload files to a temporary directory. You can choose the directory using the SecUploadDir directive:

```
SecUploadDir /tmp
```

Verifying files

You can choose to execute an external script to verify a file before it is allowed to go through the web server to the application. The following line will turn this feature on:

```
SecUploadApproveScript /full/path/to/the/script.sh
```

The script will be given one parameter on the command line - the full path to the file being uploaded. It may do with the file whatever it likes. After processing it, it should write the response on the standard output. If the first character of the response is "1" the file will be accepted. Anything else, and the whole request will be rejected.

Storing uploaded files

You can choose to keep files uploaded through the web server. Simply do the following:

```
SecUploadKeepFiles On
```

Files will be stored at a path defined using the SecUploadDir directive.

Upload memory limit

Apache 1.x does not offer proper infrastructure to intercept requests. Since what mod_security is doing is a hack, there is no way (none that I know, at least) to store the request anywhere else but in memory. With Apache 1.x there is a choice to analyse

multipart/form-data (upload) requests in memory or not analyse them at all (selectively turn POST processing off).

With Apache 2.x, however, you can define the amount of memory you want to spend parsing multipart/form-data requests in memory. If a request is larger than the memory you have allowed a temporary file will be used. The default value is 60 KB.

```
SecUploadInMemoryLimit 125000
```

Other features

Server identity masking

One technique that often helps slow down and confuse attackers is the web server identity switch. Web servers typically send their identity with every HTTP response in the Server: header. Apache is particularly helpful here, not only sending its name and full version by default, but it also allows server modules to append their versions too.

To change the identity of the Apache web server you would have to go into the source code, find where the name "Apache" is hardcoded, change it, and recompile the server. As of 1.7 you can do this at runtime with a mod_security configuration option:

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

It should be noted that although this works quite well, skilled attackers (and tools) may use other techniques to "fingerprint" the web server. For example, default files, error message, ordering of the outgoing headers, the way the server responds to certain requests and similar - can all give away the true identity. I will look into further enhancing the support for identity masking in the future releases of mod_security.

If you change Apache signature but you are annoyed by the strange message in the error log (some modules are still visible - this only affects the error log, from the outside it still works as expected):

```
[Fri Jun 11 04:02:28 2004] [notice] Microsoft-IIS/5.0 mod_ssl/2.8.12
OpenSSL/0.9.6b configured -- resuming normal operations
```

Then you should re-arrange the modules loading order to allow mod_security to run last, exactly as explained for chrooting.

Note: In order for this directive to work you must leave/set ServerTokens to "Full".

Chroot support

Standard approach

Starting with v1.5.1 mod_security includes support for Apache chrooting (v1.6 for Apache 2.x). Chrooting is a process of confining an application into a special part of the file system, sometimes called the "jail". Once the chroot call is performed, the application can no longer access what lies outside the jail. Only the root user can

escape the jail, and a vital part of the chrooting process is not allowing anything root related (root processes or suid root binaries) inside the jail. The idea is that if an attacker manages to break in through the web server he won't have much to do because he, too, will be in jail, with no means to escape.

Applications do not have to support chrooting. Any application can be chrooted using the chroot binary. The following line:

```
chroot /chroot/apache /usr/local/web/bin/apachectl start
```

will start Apache but only after replacing the file system with what lies beneath /chroot/apache.

Unfortunately, things are not as simple as this. The problem is that applications typically require shared libraries, various files, and other binaries to function properly. So, to make them function you must make copies of required files and make them available inside the jail. This is no easy task (take a look at http://penguin.epfl.ch/chroot.html for detailed instructions on how to chroot an Apache web server).

The mod_security way

While I was chrooting an Apache the other day I realized that I was bored with the process and I started looking for ways to simplify it. As a result, I built the chrooting functionality into the mod_security module itself, making the whole process less complicated. With mod_security under your belt, you only need to add one line to the configuration file:

```
SecChrootDir /chroot/apache
```

and your web server will be chrooted successfully.

Apart from simplicity, mod_security chrooting brings another advantage. Unlike external chrooting (mentioned previously) mod_security chrooting requires no additional files to exist in jail. The chroot call is made after web server initialization but before forking. Because of this, all shared libraries are already loaded, all web server modules are initialized, and log files are opened. You only need your data in jail.

There are some cases, however, when you will need additional files in jail, and that is if you intend to execute CGI scripts or system binaries. They may have their own file requirements. If you fall within this category then you need to proceed with the external chroot procedure as normal but you still won't have to think of the Apache itself.

Note: With Apache 2.x, the default value for the AcceptMutex directive is "pthread". Sometimes this setting prevents mod_security from working. Set AcceptMutex to any other setting to overcome this problem (e.g. "posixsem").

Note: If you configure chroot to leave log files outside the jail, Apache will have file descriptors pointing to files outside the jail. The chroot mechanism was not initially

designed for security and some people fill uneasy about this. Make your own decision. **Treat this feature as somewhat experimental.** One more thing to note is that files required for authentication must be in the jail too (Apache opens them on every request).

Required module ordering for chroot support (Apache 1.x)

As mentioned above, the chroot call must be performed at a specific moment in Apache initialization, only after all other modules are initialized. This means that mod_security must be first on the list of modules. To ensure that, you will probably need to make some changes to module ordering, using the following configuration directives:

```
ClearModuleList
AddModule mod_security.c
AddModule ...
AddModule ...

AddModule ...
```

The first directive clears the list. You must put mod_security next, followed by all other modules you intend to use (except http_core.c, which is always automatically added and you do not have to worry about it). You can find out the list of built-in modules by executing the httpd binary with the -l switch:

```
./httpd -l
```

Note: If you choose to put the Apache binary and the supporting files outside of jail, you won't be able to use the "apachectl graceful" and "apachectl restart" commands anymore. That would require Apache reaching out of the jail, which is not possible. With Apache 2, even the "apachectl stop" command does not work. For future releases I will create replacement scripts to work around this problem.

Required module ordering for chroot support (Apache 2.x)

With Apache 2.x you shouldn't need to manually configure module ordering since Apache 2.x already includes support for module ordering internally. ModSecurity uses this feature to tell Apache 2.x when exactly to call it and chroot works (if you're having problems let me know).

Note: There was a change in how the process is started in Apache2. The httpd binary itself now creates the pid file with the process number. Because of this you will need to put Apache in jail at the same folder as outside the jail. Assuming your Apache outside jail is in

```
/usr/local/web/apache
```

and you want jail to be at

```
/chroot
```

you must create the folder

```
/chroot/usr/local/web/apache/logs
```

When started, the Apache will create its pid file there (assuming you haven't changed the position of the pid file in the httpd.conf in which case you probably know what you're doing).

How mod_security chroot works

If you encounter problems on your platform it may be useful to know how mod_security performs the chroot.

Module ordering is necessary because we don't change the source code of the server. The other problem we need to overcome is the fact that all modules are initialized twice. This is a problem because, in the initialization function, we can't tell whether we are being called for the first or for the second time. Prior to 1.8dev2 mod_security used the PID of the parent process - we thought it was always set to 1 during the second initialization call. Unfortunately, this is not always the case. Starting with 1.8dev2 mod_security uses a lock file which it creates during the first initialization phase and erases it in the second go.

By default, the file is created in the logs folder, relative to the web server root "logs/modsec_chroot.lock". Use the SecChrootLock directive to change it to some other path.

Note: Also since 1.8dev2, if mod_security fails to perform chroot for any reason it will prevent the server from starting. If it fails to detect chroot failure during the configuration phase and then detects it at runtime, it will write a message about that in the error log and exit the child. This may not be pretty but it is better than running without a protection of a chroot jail when you think such protection exists.

Solving common security problems

As an example of mod_security capabilities we will demonstrate how you can use it to detect and prevent the most common security problems. We won't go into detail here about problems themselves but a very good description is available in the Open Web Application Security Project's guide, available at http://www.owasp.org.

Directory traversal

If your scripts are dealing with the file system then you need to pay attention to certain meta characters and constructs. For example, a character combination "../" in a path is a request to go up one directory level.

In normal operation there is no need for this character combination to occur in requests and you can forbid them with the following filter:

```
SecFilter "\.\./"
```

Cross site scripting attacks

Cross site scripting attacks (XSS) occur when an attacker injects HTML or/and Javascript code into your Web pages and then that code gets executed by other users. This is usually done by adding HTML to places where you would not expect them. A successful XSS attack can result in the attacker obtaining the cookie of your session and gaining full access to the application!

Proper defense against this attack is parameter filtering (and thus removing the offending HTML/Javascript) but often you must protect existing applications without changing them. This can be done with one of the following filters:

```
SecFilter "<[[:space:]]*script"
SecFilter "<.+>"
```

The first filter will protect only against Javascript injection with the "<script>" tag. The second filter is more general, and disallows any HTML code in parameters.

You need to be careful when applying filters like this since many application want HTML in parameters (e.g. CMS applications, forums, etc). You can this with selective filtering. For example, you can have the second filter from above as a general site-wide rule, but later relax rules for a particular script with the following code:

```
<Location /cms/article-update.php>
    SecFilterInheritance Off
    # other filters here ...
    SecFilterSelective "ARGS|!ARG_body" "<(.|\n)+>"
</Location>
```

This code fragment will only accept HTML in a named parameter "body". In reality you will probably add a few more named parameters to the list.

SQL/database attacks

Most Web applications nowadays rely heavily on databases for data manipulation. Unless great care is taken to perform database access safely, an attacker can inject arbitrary SQL commands directly into the database. This can result in the attacker reading sensitive data, changing it, or even deleting it from the database altogether.

Filters like:

```
SecFilter "delete[[:space:]]+from"
SecFilter "insert[[:space:]]+into"
SecFilter "select.+from"
```

can protect you from most SQL-related attacks. These are only examples, you need to craft your filters carefully depending on the actual database engine you use.

Operating system command execution

Web applications are sometimes written to execute operating system commands to

perform operations. A persistent attacker may find a hole in the concept, allowing him to execute arbitrary commands on the system.

A filter like this:

```
SecFilterSelective ARGS "bin/"
```

will detect attempts to execute binaries residing in various folders on a Unix-related operating system.

Buffer overflow attacks

Buffer overflow is a technique of overflowing the execution stack of a program and adding assembly instructions in an attempt to get them executed. In some circumstances it may be possible to prevent these types of attack by using the line similar to:

```
SecFilterByteRange 32 126
```

as it will only accept requests that consists of bytes from this range. Whether you use this type of protection or not depends on your application and the used character encoding.

If you want to support multiple ranges, regular expressions come to rescue. You can use something like:

```
SecFilterSelective THE_REQUEST "!^[\x0a\x0d\x20-\x7f]+$"
```

Custom logging

Since 1.8dev2 It is possible to use Apache custom logging to log only those requests where mod_security was involved. This is because mod_security now defines an environment variable mod_security-relevant whenever it performs an action. To have a custom log, use this:

```
CustomLog logs/modsec_custom_log \
"%h %l %u %t \"%r\" %>s %b %{mod_security-message}i" \
env=mod_security-relevant
```

Audit logging

Standard Apache logging will not help much if you need to trace back steps of a particular user or an attacker. The problem is that only a very small subset of each request is written to a log file. This problem can be remedied with the audit logging feature of mod_security. These two directives:

```
SecAuditEngine On
SecAuditLog logs/audit_log
```

will let mod_security know that you want a full audit log stored into the log file audit log. Here is an example of how a request is logged:

```
=======================================
Request: 192.168.0.2 - - [[18/May/2003:11:20:43 +0100]] "GET /cgi-
bin/printenv?p1=666 HTTP/1.0" 406 822
Handler: cgi-script
---------------------------------------
GET /cgi-bin/printenv?p1=666 HTTP/1.0
Host: wkx.dyndns.org:8080
User-Agent: mod_security regression test utility
Connection: Close
mod_security-message: Access denied with code 406. Pattern match "666" at
ARGS_SELECTIVE.
mod_security-action: 406

HTTP/1.0 406 Not Acceptable
=======================================
```

You can see that on the first line you get what you normally get from Apache. The second line contains the name of the handler that was supposed to handle the request. Full request (with additional mod_security headers) is given after the separator, and the response headers (in this case there is only one line) is given after one empty line.

When POST filtering is on, the POST payload will always be included in the audit log. Actual response will never be included (at least not in this version).

At this time, the audit logging part of the module will log Apache 1.x error messages, on the line below the "Handler:" line. The line will always begin with "Error:". This functionality will be added to the Apache 2.x version of the module if possible.

Unique request identifiers

If you add mod_unique_id to the Apache configuration mod_security will detect it and use the environment variable it generates (UNIQUE_ID). Its value will be written to the audit log. You could write the unique ID in an error page to the user and use it later to track and fix a false positive.

Choosing what to log

Starting with v1.5.1 the SecAuditEngine parameter accepts one of four values:

- On – log all requests
- Off – do not log requests at all
- RelevantOnly – only log relevant requests. Relevant requests are those requests that caused a filter match.
- DynamicOrRelevant – log dynamically generated or relevant requests. A request is considered dynamic if its handler is not null.

Getting mod_security to log dynamic requests could require a little bit of work. In theory, a response to a request is generated by a handler, and if there is a handler

attached to a request it can be considered to be of a dynamic nature. In practice, however, Apache can be configured to server dynamic pages without a handler (it will choose the module based on the mime type). This will happen, for example, if you configure PHP as instructed in the main distribution:

```
AddType application/x-httpd-php .php
```

While this works, it won't serve our purpose here. However, if you replace the above line with the following:

```
AddHandler application/x-httpd-php .php
```

PHP will work just as well and audit logger will be able to do its job.

The testing utility

A small testing utility was developed as part of the mod_security effort. It provides a simple and easy way to send crafted HTTP requests to a server, and to determine whether the attack was successfully detected or not.

Calling the utility without parameters will result in its usage printed:

```
[ivanr@wkx tests]$ ./run-test.pl
Usage: ./run-test.pl host[:port] testfile1, testfile2, ...
```

First parameter is the hostname of the server, with port being optional. All other parameters are filenames of files containing crafted HTTP requests.

To make your life a little bit easier, the utility will generate certain request headers:

- Host: (hostname)
- User-Agent: mod_security regression testing utility
- Connection: Close

You can include them in the request if you want, the utility will not add them if they are already there.

Here is how an HTTP request looks like:

```
# 01 Simple keyword filter
#
# mod_security is configured not to allow
# the "/cgi-bin/keyword" pattern
#
GET /cgi-bin/keyword HTTP/1.0
```

This request consists only of the first line, with no additional headers. You can create as complicated requests as you wish. Here is one example of a POST method usage:

```
# 10 Keyword in POST
#
```

```
POST /cgi-bin/printenv HTTP/1.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 5

p=333
```

Lines that are at the beginning of the file and begin with # will be treated as comments. The first line is special, and it should contain the name of the test.

The utility expects status 200 as a result and will treat such responses as successes. If you want some other response you need to tell it by writing the expected response code on the first line (anywhere on the line). Like this:

```
# 14 Redirect action (requires 302)
#
GET /cgi-bin/test.cgi?p=xxx HTTP/1.0
```

The brackets and the "requires" keyword are not required but are recommended for better readability.

Technology specific notes

PHP

Preventing register_global problems

Nowadays it is widely accepted that using the register_globals feature of PHP leads to security problems, but it wasn't always like this (if you don't know what this feature is then you are probably not using it; but, hey, read on the discussion is informative). In fact, the register_globals feature was turned on by default until version 4.2.0. As a result of that, many applications that exist depend on this feature (for more details have a look at http://www.php.net/register_globals).

If you can choose, it is better to refactor and rewrite the code to not use this feature. But if you cannot afford to do that for some reason or another, you can use mod_security to protect an application from a known vulnerability. Problematic bits of code usually look like this:

```
<?php
// this is the beginning of the page
if ($authorised) {
    // do something protected
}
// the rest of the page here
?>
```

And the attacker would take advantage of this simply by adding an additional parameter to the URL. For example, http://www.modsecurity.org/examples/test.php?authorised=1

Rejecting all requests that explicitly supply the parameter in question will be

sufficient to protect the application from all attackers:

```
<Location "/vulnerable-application/">
    SecFilterSelective ARG_authorised "!$^"
</Location>
```

The filter above rejects all requests where the variable "authorised" is not empty. You can also see that we've added the <Location>...</Location> directives to limit filter only to those parts of the web server that really need it.

Additional Examples

Parameter checking

Regular expressions can be pretty powerful. Here is how you can check whether a parameter is an integer between 0 and 99999:

```
SecFilterSelective ARG_parameter "!^[0-9]{1,5}$"
```

File upload

Forbid file upload for the application as a whole, but allow it in a subfolder:

```
# Reject requests with header "Content-Type" set to "multipart/form-data"
SecFilterSelective "HTTP_CONTENT_TYPE" multipart/form-data

# Only for the script that performs upload
<Location "/upload.php">
    # Do not inherit filters from the parent folder
    SecFilterInheritance Off
</Location>
```

Securing FormMail

Earlier versions of FormMail could be abused to send email to any recipient (I've been told that there is a new version that can be secured properly).

```
# Only for the FormMail script
<Location /cgi-bin/FormMail>
    # Reject request where the value of parameter "recipient"
    # does not end with "@webkreator.com"
    SecFilterSelective "ARG_recipient" "!@webkreator.com$">
</Location>
```

Performance

The protection provided by mod_security comes at a cost. Your web server becomes a little bit slower and uses more memory to operate.

Speed

In my experience, the speed difference is not significant. I did some testing at the early stages of development and the speed difference was around 10%. However, if you configure mod_security to work only on dynamic requests the difference becomes slower - on real web sites an access to a dynamic page is accompanies by several access to other types of pages (CSS, JavaScript, images).

Memory consumption

In order to be able to analyze a request, mod_security stores it in memory. In most cases this is not a big deal since most requests are small. However, it can be a problem for parts of the web site where files are being uploaded. To avoid this problem you need to turn mod_security POST scanning off for those parts of the web site. In any case it is advisable to review and configure various limits in the Apache configuration (see http://httpd.apache.org/docs/mod/core.html#limitrequestbody for a description of LimitRequestBody, LimitRequestsFields, LimitRequestFieldsize and LimitRequestLine directives).

Other things to watch for

The debugging feature can be very useful but it writes large amounts of data to a file for every request. As such it creates a bottleneck for busy servers. There is no reason to use the debugging mode on production servers so keep it off.

The audit log feature is similar and also introduces a bottleneck for two reasons. First, large amounts of data are written to the file, and second, access to the file must be synchronized. If you still want to use the audit log try to create many different audit logs, one for each application running on the server, to minimize the synchronization overhead (this advice does not remove the overhead in the Apache 2.x version because synchronization is performed via a central mutex).

Known issues

- The Apache 2.x programming API is not well documented nor stable at this time. Consequently, mod_security for Apache 2.x cannot be more stable than the server itself. Use your own judgment.

Other resources

Other interesting resources available:

- Web Security Appliance With Apache and mod_security, a SecurityFocus article: http://www.securityfocus.com/infocus/1739
- Introduction to mod_security, published on ONLamp.com: http://www.onlamp.com/pub/a/apache/2003/11/26/mod_security.html

Appendix A: Recommended Configuration

Below is the recommended minimal mod_security configuration. It is only a starting point designed not to give you an instant headache. You should look into tightening the configuration where you can.

```
    # Only inspect dynamic requests
    # (YOU MUST TEST TO MAKE SURE IT WORKS AS EXPECTED)
    SecFilterEngine DynamicOnly

    # Reject requests with status 403
    SecFilterDefaultAction "deny,log,status:403"


    # Some sane defaults
    SecFilterScanPOST On
    SecFilterCheckURLEncoding On
    SecFilterCheckCookieFormat On
    SecFilterCheckUnicodeEncoding Off

    # Accept almost all byte values
    SecFilterForceByteRange 1 255

    # Server masking is optional
    # SecServerSignature "Microsoft-IIS/5.0"

    SecUploadDir /tmp
    SecUploadKeepFiles Off

    # Only record the interesting stuff
    SecAuditEngine RelevantOnly
    SecAuditLog logs/audit_log

    # You normally won't need debug logging
    SecFilterDebugLevel 0
    SecFilterDebugLog logs/modsec_debug_log

    # Only accept request encodings we know how to handle
    # we exclude GET requests from this because some (automated)
    # clients supply "text/html" as Content-Type
    SecFilterSelective REQUEST_METHOD "!^GET$" chain
    SecFilterSelective HTTP_Content-Type "!(^$|^application/x-www-form-
urlencoded$|^multipart/form-data;)"

    # Require Content-Length to be provided with
    # every POST request
    SecFilterSelective REQUEST_METHOD "^POST$" chain
    SecFilterSelective HTTP_Content-Length "^$"

    # Don't accept transfer encodings we know we don't handle
    # (and you don't need it anyway)
    SecFilterSelective HTTP_Transfer-Encoding "!^$"
```