ModSecurity for Apache 1.8.7 User Guide

http://www.modsecurity.org

Table of Contents

Introduction

ModSecurity is an open source intrusion detection and prevention engine for web applications. It can also be called an web application firewall. 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. (Strictly speaking, some processing is done on the request before it reaches ModSecurity but that is unavoidable in the embedded mode of operation.)
- **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. For example, it is possible to look at individual parameters, or named cookie values.
- **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 forensic analysis later.
- **HTTPS filtering**; since the engine is embedded in the web server, it gets access to request data after decryption takes place.
- **Compressed content filtering**; same as above, the security engine has access to request data after decompression takes place.

ModSecurity can be used to detect attacks, or to detect and prevent attacks.

Licensing

ModSecurity is available under two licenses. Users can choose to use the software under the terms of the GNU General Public License (http://www.gnu.org/licenses/gpl.html), as an Open Source / Free Software product. Alternatively, a variety of commercial licenses is available: end-user licenses for individual or site-wide deployment, OEM licenses for closed-source distribution with applications, web servers, or security appliances. For more information on commercial licensing please contact Thinking Stone Ltd:

Thinking Stone Ltd
Tel: +44 845 0580628
Fax: +44 870 7623934
http://www.thinkingstone.com
contact@thinkingstone.com

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 the Apache modules I used to learn Apache module programming from. My special thanks goes to those who designed and wrote *mod_rewrite*.

Contact

ModSecurity is developed by Ivan Ristic and Thinking Stone. Comments and feature requests are welcome. Please send your emails to ivanr@webkreator.com.

> Please do not send support requests to my personal email address. I do spend time responding to support queries but I don't respond privately any more. Doing so prevents other users from using mail archives to find answers for themselves. If you need answers quickly or you want guaranteed response times consider purchasing commercial support from Thinking Stone.

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 version of ModSecurity directly from CVS (best features, but possibly unstable) or use the latest stable release (**recommended**). If you choose a stable release, it might be possible to install ModSecurity from binary. It is always possible to compile it from source code.

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

CVS Access

If you want to access the latest version of the module you need to get it from the CVS repository. The list of changes made since the last stable release is normally available on the web site (and in the file *CHANGES*). The CVS repository for ModSecurity is 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 repository.

Nightly Snapshot Download

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.

Stable Release Download

To download the stable release go to http://www.modsecurity.org/download/. Binary distributions are sometimes available. If they are, they are listed on the download page. If not download the source code distribution.

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.c* into a dynamic shared object (DSO).

DSO

Installing as DSO is easier, and the procedure is the same for both Apache branches. First unpack the distribution somewhere (anywhere will do), and compile the module with:

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

After this you only need to stop and then start Apache (if you try to restart it you may get a segfault):

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

> I've had reports from people using platforms that do not have the *apxs* utility installed. In some Unix distribution this tool is distributed in a separate package. The problem arises when that package is not installed b default. To resolve this problem read the documentation from your vendor to discover how you can add your own custom Apache modules. (On some RedHat platforms you need to install the package **http-devel** to get access to the *apxs* utility).

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 (and subsequent maintenance) is a bit more complicated.

After unpacking the distribution copy *mod_security.c* to the folder *src/modules/extra* in the Apache source code tree. Then reconfigure specifying the configuration script to enable the new module:

```
# ./configure \
<your other configuration options here> \
--activate-module=src/modules/extra/mod_security \
--enable-module=security
```

Compile, install, and start the web server as you normally do.

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. When installing from binary you are likely to have two DSO libraries in the distribution, one for each major Apache branch. Choose the file appropriate for the version you are using. Then proceed as described below:

Apache 1.x

Copy *mod_security.so* (on Unix) or *mod_security.dll* (on Windows) to *libexec/* (this folder is relative to the Apache installation, not the source tree). Then add the following line to *httpd.conf*:

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

Depending on your existing configuration (you may have chosen to configure module loading order explicitly) it may be necessary to activate the module using the `AddModule` directive:

```
AddModule mod_security.c
```

In most cases it is not important where you add the line. It is recommended (and, in fact, mandatory if you intend to use the internal chroot feature) to make *mod_security* execute last in the module chain.Read the section “Required module ordering for chroot support (Apache 1.x)” for more information.

Apache 2.x

Copy *mod_security.so* (on Unix) or *mod_security.dll* (on Windows) to *modules/* (this folder is relative to the Apache installation, not the source tree). Then add the following line to *httpd.conf*:

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

Configuration

ModSecurity configuration directives are added to your configuration file (typically *httpd.conf*) directly. When it is not always certain whether module will be enabled or disabled on web server start it is customary to enclose its configuration directives in a `<IfModule>` container tag. This allows Apache to ignore the configuration directives when the module is not active.

```
<IfModule mod_security.c>
    # mod_security configuration directives
```

```
    # ...
</IfModule>
```

Since Apache allows configuration data to exist in more than one file it is possible to group ModSecurity configuration directives in a single file (e.g. *modsecurity.conf*) and include it from *httpd.conf* with the `Include` directive:

```
Include conf/modsecurity.conf
```

Turning filtering on and off

The filtering engine is disabled by default. To start monitoring requests add the following to your configuration file:

```
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 the precious CPU cycles to check requests for static files. To understand how ModSecurity decides what is a dynamically generated request read the section "Choosing what to log".

POST scanning

Request body payload (or `POST` payload) scanning is disabled by default. To use it, you need to turn it on:

```
SecFilterScanPOST On
```

mod_security supports two encoding types for the request body:

- **`application/x-www-form-urlencoded`** - used to transfer form data
- **`multipart/form-data`** – used for file transfer

Other encodings are not used by most web applications. 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` payload scanning off on per-request basis. If ModSecurity sees that an environment variable `MODSEC_NOPOSTBUFFERING` is defined it will not perform `POST` payload buffering. For example, to turn `POST` payload buffering off for file uploads use the

following:

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

The value assigned to 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 *chunked transfer encoding*. ModSecurity does not support chunked requests at this time; when a request is made with chunked encoding it will ignore the body of the request. As far as I am aware no browser uses chunked encoding to send requests. Although Apache does support this encoding for some operations many modules don't.

Left unattended this may present an opportunity for an attacker to sneak malicious payload. Add the following line to your configuration to prevent attackers to exploit this weakness:

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

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

Default action list

Whenever a rule is matched against a request, one or more actions are performed. Individual filters can each have their own actions but it is easier to define a default set of actions for all filters. (You can always have per-rule actions if you want.) You define default actions with the configuration directive `SecFilterDefaultAction`. For example, the following will configure the engine to log each rule match, and reject the request with status code `404`:

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

The `SecFilterDefaultAction` directive accepts only one parameter, a comma-separated list of actions separated. The actions you specify here will be performed on every filter match, except for rules that have their own action lists.

> As of 1.8.6, if you specify a non-fatal default action list (a list that will not cause the request to be rejected, for example `log,pass`) such action list will be **ignored** during the initialization phase. The initialization phase is designed to gather information about the request. Allowing non-fatal actions would cause some pieces of the request to be missing. Since this information is required for internal processing such actions cannot be allowed. If you want ModSecurity to operate in a "detect-only" mode you need to disable all implicit validations (URL encoding validation, Unicode encoding validation, cookie format validation, and byte range restrictions).

Implicit validation

As of 1.8.6 implicit request validation (if configured) will be performed only at the beginning of request processing. After that only will normalization be performed. This change allows for better protection (request headers are now automatically checked at the beginning). It also allows for a "detect-only" mode of operation of rules in the rule set.

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 the time. Sometimes you need to relax checks in some part of the site. By using the `SecFilterInheritance` directive:

```
SecFilterInheritance Off
```

you can instruct ModSecurity to disregard parent filters so that you can start with rules from the scratch. This directive affects **only** rules. The configuration is always inherited from the parent context but you can override it as you are pleased using appropriate configuration directives.

> Configuration and rule inheritance is **always** enabled by default. If you have a configuration context beneath one that has had inheritance disabled you will have to explicitly disable inheritance again if that is what you need.

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` represents 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. ModSecurity checks all supplied encodings in order to verify they are valid.

You can turn URL encoding validation on with the following line:

```
SecFilterCheckURLEncoding On
```

> This directive does not check encoding in a POST payload when the `multipart/form-data` encoding (file upload) is used. It is not necessary to do so because URL encoding is not used for this encoding.

Unicode Encoding Validation

Like many other features Unicode encoding validation is disabled by default. You should turn it on if your application or the underlying operating system accept/understand Unicode.

> 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.

> This directive does not check byte range in a POST payload when `multipart/form-data` encoding (file upload) is used. Doing so would prevent binary files from being uploaded. However, after the parameters are extracted from such request they are checked for a valid range.

Allowing others to see ModSecurity

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 the response code `500`, which stands for “Internal Server Error”. Attackers that encounter such a response might think that your application has crashed. Since vulberable applications often crash when they encounter unexpected input the fact ModSecurity is active may remain hidden from the attacker.

There is another school of thought on this matter, which says that you should not hide the fact that you are running ModSecurity. 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. ModSecurity keeps quiet buy default (this is the recommended state), 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 how you configured ModSecurity.

Debugging

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

```
SecFilterDebugLog logs/modsec_debug_log
```

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

```
SecFilterDebugLevel 4
```

Possible log values are:

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

> ModSecurity uses log levels up until 9 internally but they are only useful for debugging purposes.

Request filtering

When the filtering engine is enabled, every incoming request is intercepted and analysed before it is processed. The analysis begins with a series of built-in checks designed to validate the request format. These checks can be controlled using configuration directives. In the second stage, the request goes through 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 is, well, simple. It looks like this:

```
SecFilter KEYWORD
```

For each simple filter like this, ModSecurity 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 of the request will be searched too (provided the request body buffering is enabled, of course).

Path normalization

Filters are not applied to raw request data, but on a normalized copy instead. 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 a string `/bin/./sh` (which has the same meaning) in order to avoid the filter.

ModSecurity automatically applies the following transformations:

- On Windows only, convert \ to `/`
- Reduce `/./` to `/`
- Reduce // to /
- Decode URL-encoded characters

You can choose whether to enable or disable the following checks:

- Verify URL encoding
- Allow only bytes from a certain range to be used

> Normalization, as implemented in the current version of ModSecurity, can sometimes make your job harder. For example you may have a hard time writing a signature that will detect a URI somewhere. This is because a URI that appears in the request as "http://" will be translated during the normalisation process to "http:/". Note that there is only one forward slash in the second version.

Null byte attack prevention

Null byte attacks try to confuse C/C++ based software and trick it into thinking that a string ends before it actually does. This type of an attack is typically rejected with a proper `SecFilterByteRange` filter. However, if you do not do this a null byte can interfere with ModSecurity processing. To fight this, ModSecurity 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:

```
GET /one/two/three?p=visible%00hidden HTTP/1.0
```

it now works as expected.

Regular expressions

The simplest method of filtering I discussed earlier is actually slightly more complex. Its full syntax is as follows:

```
SecFilter KEYWORD [ACTIONS]
```

First of all, the keyword is not a simple text. It is an regular expression. A regular expression is a mini programming language designed to pattern matching in text. 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

> 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 (for example. Windows). Stack overflow will result in non-exploitable segmentation fault.

The second parameter is an action list definition, which specifies what will happen if the filter matches. Actions are explained later in this manual.

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. Examine the following example:

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

It will apply the regular expression only the IP address of the client and the host name.

The list of possible location identifiers 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

ModSecurity provides full support for Cookies. By default cookies will be treated as if they were in version 0 format (Netscape-style cookies). However, version 1 cookies (as defined in RFC 2965) are also supported. To enable version 1 cookie support use the SecFilterCookieFormat directive:

```
# enable version 1 (RFC 2965) cookies
SecFilterCookieFormat 1
```

By default, ModSecurity will not try to normalize cookie names and values. However, since some applications and platforms (e.g. PHP) do encode cookie content you can choose to apply normalisation techniques to cookies. This is done using the `SecFilterNormalizeCookies` directive.

```
SecFilterNormalizeCookies On
```

> Prior to version 1.8.7 ModSecurity supported the `SecFilterCheckCookieFormat` directive. Due to recent changes in 1.8.7 this directive is now deprecated. It can still be used in the configuration but it does not do anything. The directive will be completely removed in the 1.9.x branch.

Output filtering

ModSecurity supports output filtering in the version for Apache 2. It is disabled by default so you need to enable it first:

```
SecFilterScanOutput On
```

After that, simply add selective filters using a special 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. The following will catch PHP output error in the response body, replace the response with an error, and execute a custom PHP script (so that the application administrator can be notified):

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

You should note that although you can mix output filters with input filters, they are **not** executed at the same time. Input filters are executed before a request is processed by Apache, while the output filters are executed after Apache completes request processing.

> Actions `skipnext` and `chain` do not work with output filters.

Output filtering is only useful for plain text and HTML output. Applying regular expressions to binary content (for example images) will only slow down the server. ModSecurity can selectively apply output filtering to responses based on their mime type. Using the `SecFilterOutputMimeTypes` directive you can tell it which mime types to watch for:

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

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

> Using output buffering will make ModSecurity 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 length.

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. For example, `exec` is one secondary action.
- Flow actions can change the flow of rules, causing the filtering engine to jump to another rule, or to skip one or several rules. Flow actions are `chain` and `skip`.

- 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 most filter matches:

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

This example defines an action list that consists of three actions. Commas are used to separate actions in a list. The first two actions consist of a single word. But the 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 (`SecFilter` and `SecFilterSelective`) accept a set of actions as an optional parameter. When using per-filter actions it is assumed the request will be rejected on a filter match. If you want to allow the request to proceed you must do that by explicitly setting a non-fatal action (such as “log,pass”.

> As of 1.8.6, if you specify a non-fatal default action (such as "log,pass") then it will be **ignored** during mod_security initialization phase. The initialization phase is designed to gather information about the request, allowing non-fatal actions would cause some pieces of the request to be missing (for internal processing in mod_security). Therefore if you want mod_security to operate in a "detect-only" mode you should disable all implicit validations (check URL encoding, Unicode, cookie format, byte range).

Built-in actions

pass

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

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

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:

```
# stop filter processing for request coming from
# the administrator's workstation
SecFilterSelective REMOTE_ADDR "^192.168.2.99$" allow
```

deny

Interrupt request processing on a filter match. Unless the `status` action is used too, ModSecurity will immediately return a HTTP `500` error code. If a request is denied the header `mod_security-action` will be added to the list of request headers. This header will contain the status code used.

status

Use the supplied HTTP status code when request is denied. The following rule:

```
SecFilter KEYWORD "deny,status:404"
```

will return a "Page not found" response when triggered. The Apache `ErrorDocument` directive will be triggered if present in the configuration. Therefore if you have previously defined a custom error page for a given status then it will be executed and its output presented to the user.

redirect

On filter match 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. The URL must not contain a comma.

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 a primary action if it exists. 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 the header `mod_security-executed` to the list of request headers.

> You should be aware that forking a threaded process results in all threads being replicated in the new process. Forking can therefore incur larger overhead in multithreaded operation.

log

Log filter match to the Apache error log.

nolog

Do not log filter match to the 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.

> 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, ModSecurity will add information to the request headers, thus allowing your scripts to find and use them. Obviously, you will have to configure ModSecurity 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 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 configuration returns a HTTP status code `500`, and you configure Apache to execute a custom script whenever this code occurs (for example: `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_*` (as described here: http://httpd.apache.org/docs-2.0/custom-error.html).

Making ModSecurity 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 (I have written a helper script that works with `iptables`, download it from here: http://www.apachesecurity.net).

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 requests from a proxy server can be very dangerous since all legitimate users will be affected 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

ModSecurity supports the `multipart/form-data` encoding used for file uploads.

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
```

It is better to choose a private directory for file storage, somewhere only the web server user is allowed access. Otherwise, other server users may be able to access the files uploaded through the web server.

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 `SecUploadApproveScript` directive enables this function. Like in the following example:

```
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. Your script may use the rest of the line to write a more descriptive error message. This message will be stored to the debug log.

Storing uploaded files

You can choose to keep files uploaded through the web server. Simply add the following line to your configuration:

```
SecUploadKeepFiles On
```

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

Upload memory limit

Apache 1.x does not offer a proper infrastructure for request interception. It is only possible to intercept requests storing them completely in the operating memory. With Apache 1.x there is a choice to analyse `multipart/form-data` (file 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. When a request is larger than the memory you have allowed a temporary file will be used. The default value is 60 KB but the limit can

be changed using the `SecUploadInMemoryLimit` directive:

```
SecUploadInMemoryLimit 125000
```

Impedance mismatch

Web application firewalls have a difficult job trying to make sense of data that passes by, without any knowledge of the application and its business logic. The protection they provide comes from having an independent layer of security on the outside. Because data validation is done twice, security can be increased without having to touch the application. In some cases, however, the fact that everything is done twice brings problems. Problems can arise in the areas where the communication protocols are not well specified, or where either the device or the application do things that are not in the specification.

The worst offender is the cookie specification. (Actually all four of them: http://wp.netscape.com/newsref/std/cookie_spec.html, http://www.ietf.org/rfc/rfc2109.txt, http://www.ietf.org/rfc/rfc2964.txt, http://www.ietf.org/rfc/rfc2965.txt.) For many of the cases, possible in real life, there is no mention in the specification - leaving the programmers to do what they think is appropriate. For the largest part this is not a problem when the cookies are well formed, as most of them are. The problem is also not evident because most applications parse cookies they themselves send. It becomes a problem when you think from a point of view of a web application firewall, and a determined adversary trying to get past it. I'll explain with an example.

In the 1.8.x branch and until 1.8.6 ModSecurity (I made improvements in 1.8.7) used a v1 cookie parser. When I wrote the parser I thought it was really good because it could handle both v0 and v1 cookies. However, I made a mistake of not thinking like an attacker would. As Stefan Esser pointed out to me, the differences between v0 and v1 formats could be exploited to make a v1 parser see one cookie where a v0 parser would see more. Here it is:

```
Cookie: innocent="; nasty=payload; third="
```

You see, a v0 parser does not understand double quotes. It typically only looks for semi-colons and splits the header accordingly. Such a parser sees cookies "innocent", "nasty", and "third". A v1 parser, on the other hand, sees only one cookie - "innocent".

How is the impedance mismatch affecting the web application firewall users and developers? It certainly makes our lives more difficult but that's all right - it's a part of the game. Developers will have to work to incorporate better and smarter parsing routines. For example, there are two cookie parsers in ModSecurity 1.8.7 and the user can choose which one to use. (A v0 format parser is now used by default.) But such improvements, since they cannot be automated, only make using the firewall more difficult - one more thing for the users to think about and configure.

On the other hand, the users, if they don't want to think about cookie parsers, can always fall back to use those parts of `HTTP` that are much better defined. Headers, for example. Instead of using `COOKIE_innocent` to target an individual cookie they can just use `HTTP_Cookie` to target the whole cookie header. Other variables, such as `ARGS`, will look at all variables at once no matter how hard adversaries try to mask them.

Other features

Server identity masking

One technique that often helps slow down and confuse attackers is the web server identity change. 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 hard-coded, change it, and recompile the server. The same effect can be achieved using the `SecServerSignature` directive:

```
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.

> In order for this directive to work you must leave/set `ServerTokens` to `Full`.

Chroot support

Standard approach

ModSecurity includes support for Apache filesystem isolation, or chrooting. Chrooting is a process of confining an application into a special part of the file system, sometimes called a "jail". Once the chroot (short for “change root”) operation is performed, the application can no longer access what lies outside the jail. Only the *root* user can escape the jail. A vital part of the chrooting process is not allowing anything *root* related (*root* processes or *root suid* 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, and various other files and 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 not an 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.

With Apache 2.x, the default value for the `AcceptMutex` directive is `pthread`. Sometimes this setting prevents Apache from working when the chroot functionality is used. Set `AcceptMutex` to any other setting to overcome this problem (e.g. `posixsem`).

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.**

The files used by Apache for authentication must be inside the jail since these files are opened 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 ModSecurity must be the 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 ModSecurity 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
```

> 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 may 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).

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 a 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 the mod_security chroot works

If you encounter problems on your platform it may be useful to know how mod_security performs the chroot isolation. 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. (Actually, it is possible to

determine that in Apache 2 but not in Apache 1.) ModSecurity 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.

> Since version 1.8, if ModSecurity 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 ModSecurity 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 "<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" "<.+>"
</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 1.8 it is possible to use Apache custom logging to log only those requests where mod_security was involved. This is because ModSecurity now defines an environment variable `mod_security-relevant` whenever it performs an action. To use a custom log file, add the following (or similar) to your configuration:

```
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 ModSecurity. 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 the `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

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 ModSecurity to log dynamic requests can sometimes require a little bit of work depending on your configuration. In Apache theory, a response to a request is generated by a so-called *handler*. If there is a handler attached to a request it should be considered to be of a dynamic nature. In practice, however, Apache can be configured to server dynamic pages without a handler (it then chooses the module based on the resource 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 isn't entirely correct. However, if you replace the above line with the following:

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

PHP will work just as well, Apache will have a handler assigned to the request, and audit logger will be able to log selectively.

The testing utility

A small HTTP testing utility was developed as part of the ModSecurity 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:

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

First parameter is the host name 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 automatically:

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

You can include them in the request if you need to. 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 peculiarities

When writing ModSecurity rules that are meant to protect PHP applications one needs to have a list of PHP peculiarities in mind. It is often easy to design a rule that works when you are attacking yourself in one way but completely miss an attack variant. Below is a list of things I am aware about:

- When the `register_globals` is set to `On` request parameters become global variables. (In PHP 4.x it is even possible to override the `GLOBALS` array).
- Cookies are treated as request parameters.
- Whitespace at the beginning of parameters is ignored.
- The remaining whitespace (in parameter names) is converted to underscores.
- The order in which parameters are taken from the request and the environment is `EGPCS` (environment, get, post, cookies, built-in variables). This means that a `POST` parameter will overwrite the parameters transported on the request line (in `QUERY_STRING`).
- When the `magic_quotes_gpc` is set to `On` PHP will use backslash to escape the following characters: single quote, double quote, backslash, and `NULL`.
- If `magic_quotes_sybase` is set to On only the single quote will be escaped using another single quote. In this case the `magic_quotes_gpc` setting becomes irrelevant.

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 ModSecurity comes at a cost. Your web server becomes a little bit slower and uses more memory.

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 ModSecurity to work only on dynamic requests the difference becomes smaller. On real-life web sites one access to a dynamic page is accompanies by several access to other types of files (CSS, JavaScript, images). The performance impact is directly related to the complexity of the configuration. You can use the performance measurement improvements in the Apache 2 version of the module to measure exactly how much time ModSecurity spends working on each request. In my tests this was usually 2-4 milliseconds (on a server with a 2 GHz processor).

Memory consumption

In order to be able to analyze a request, ModSecurity stores the request data 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 the request body buffering off for those parts of the web site. (This is only a problem in the Apache 1.x version. The Apache 2.x version will use a temporary file on disk for storage when a request is too large to be stored in memory.) 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 disk, 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

There are some known issues:

- (Apache 2 only) ModSecurity supports response body interception but it does not take care of response headers explicitly. It may happen that some of the original response headers get through.
- (Apache 2 only) Response body interception is not fully compatible with mod_deflate at the moment. To ensure smooth operation ModSecurity will remove prevent mod_deflate from operating on those (and only on those) responses that have been intercepted.

Important notes

Please read the following notes:

- You should carefully consider the impact of every filtering rule you add to the configuration. You particularly don't want to deny access using very broad rules. This results in false positives and very angry users.
- Although ModSecurity can be used in *.htaccess* files (`AllowOverride Options` is required to do this), it **should not** be enabled for use by parties you do not trust.

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 Off
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|HEAD)$" chain
SecFilterSelective HTTP_Content-Type \
"!(^application/x-www-form-urlencoded$|^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 "!^$"
```