ModSecurity Reference Manual

Version 2.0.3 / (October 26, 2006)

Table of Contents

Introduction

ModSecurity™ is an embeddable web application firewall. It provides protection from a range of attacks against web applications and allows for HTTP traffic monitoring and real-time analysis with no changes to existing infrastructure.

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. A range of commercial licenses is also available, together with a range of commercial support contracts. For more information on commercial licensing please contact Breach Security.

Note

ModSecurity and mod_security are trademarks of Thinking Stone.

Installation

ModSecurity installation consists of the following steps:

1. ModSecurity 2.x works with Apache 2.0.x or better.
2. Make sure you have `mod_unique_id` installed.
3. (Optional) Install the latest version of libxml2, if it isn't already installed on the server.
4. Unpack the ModSecurity archive
5. Edit Makefile to configure the path to Apache (for example: `top_dir = /usr/local/apache2`).
6. (Optional) Edit Makefile to enable ModSecurity to use libxml2 (uncomment line `DEFS = -DWITH_LIBXML2`) and configure the include path (for example: `INCLUDES=-I/usr/include/libxml2`)
7. Compile with `make`
8. Stop Apache
9. Install with `make install`
10. (Optional) Add one line to your configuration to load libxml2: `LoadFile /usr/lib/libxml2.so`
11. Add one line to your configuration to load ModSecurity: `LoadModule security2_module modules/mod_security2.so`
12. Configure ModSecurity
13. Start Apache
14. You now have ModSecurity 2.x up and running.

Note

If you have compiled Apache yourself you might experience problems compiling ModSecurity against PCRE. This is because Apache bundles PCRE but this library is also typically provided by the operating system. I would expect most (all) vendor-packaged Apache distributions to be configured to use an external PCRE library (so this should not be a problem).

You want to avoid Apache using the bundled PCRE library and ModSecurity linking against the one provided by the operating system. The easiest way to do this is to compile Apache against the PCRE library provided by the operating system (or you can compile it against the latest PCRE version you downloaded from the main PCRE distribution site). You can do this at configure time using the `--with-pcre` switch. If you are not in a position to recompile Apache then, to compile ModSecurity successfully, you'd still need to have access to the bundled PCRE headers (they are available only in the Apache source code) and change the include path for ModSecurity (as you did in step 7 above) to point to them.

Do note that if your Apache is using an external PCRE library you can compile ModSecurity with `WITH_PCRE_STUDY` defined, which would possibly give you a slight performance edge in regu-

---

lar expression processing.

Configuration Directives

SecAction

Unconditionally processes the action list it receives as the first and only parameter. It accepts one parameter, the syntax of which is identical to the third parameter of `SecRule`.

SecArgumentSeparator

Specifies which character to use as separator for `application/x-www-form-urlencoded` content. Defaults to `&`. Applications are sometimes (very rarely) written to use a semicolon (`;`).

SecAuditEngine

Configures the audit logging engine. Possible values are:

- `On` - log all transactions by default.
- `Off` - do not log transactions by default.
- `RelevantOnly` - by default only log transactions that have triggered a warning or an error, or have a status code that is considered to be relevant (see `SecAuditLogRelevantStatus`).

SecAuditLog

Path to the main audit logging file. This file will be used to store the audit log entries if serial audit logging format is used. If concurrent audit logging format is used this file will be used as an index, and contain a record of all audit log files created.

Note

This file is open on startup when the server typically still runs as *root*. You should not allow non-root users to have write privileges for this file or for the directory it is stored in.

SecAuditLogParts

Default `ABCFHZ`.

Available audit log parts:

- `A` – audit log header (mandatory)
- `B` – request headers
- `C` – request body (present only if the request body exists and ModSecurity is configured to intercept it)
- `D` - RESERVED for intermediary response headers, not implemented yet.
- `E` – intermediary response body (present only if ModSecurity is configured to intercept response

bodies, and if the audit log engine is configured to record it). Intermediary response body is the same as the actual response body unless ModSecurity intercepts the intermediary response body, in which case the actual response body will contain the error message (either the Apache default error message, or the ErrorDocument page).

- `F` – final response headers (excluding the Date and Server headers, which are always added by Apache in the late stage of content delivery).
- `G` – RESERVED for the actual response body, not implemented yet.
- `H` - audit log trailer
- `I` - This part is a replacement for part C. It will log the same data as C in all cases except when `multipart/form-data` encoding in used. In this case it will log a fake `application/x-www-form-urlencoded` body that contains the information about parameters but not about the files. This is handy if you don't want to have (often large) files stored in your audit logs.
- `J` - RESERVED. This part, when implemented, will contain information about the files uploaded using multipart/form-data encoding.
- `Z` – final boundary, signifies the end of the entry (mandatory)

Note

At this time ModSecurity does not log response bodies of stock Apache responses (e.g. `404`), or the `Server` and `Date` response headers.

SecAuditLogRelevantStatus

Configures which response status code is to be considered relevant for the purpose of audit logging. The parameter is a regular expression.

SecAuditLogStorageDir

Configures the storage directory where concurrent audit log entries are to be stored. It must be writable by the web server user as new files are generated at runtime.

SecAuditLogType

Possible values are:

1. `Serial` - all audit log entries will be stored in the main audit logging file. This is more convenient for casual use but it is slower as only one audit log entry can be written to the file at any one file.
2. `Concurrent` - audit log entries will be stored in a file each.

SecChrootDir

Configures the directory path that will be used to jail the web server process.

SecCookieFormat

Selects the cookie format that will be used in the current configuration context. Possible values are:

- `0` - use version 0 (Netscape) cookies. This is what most applications use. It is the default value.
- `1` - use version 1 cookies.

SecDataDir

Path where persistent data (e.g. IP address data, session data, etc) is to be stored. Must be writable by the web server user.

SecDebugLog

Path to the debug log.

SecDebugLogLevel

Possible values are:

- `0` - no logging.
- `1` - errors (intercepted requests) only.
- `2` - warnings.
- `3` - notices.
- `4` - details of how transactions are handled.
- `5` - as above, but including information about each piece of information handled.
- `9` - log everything, including very detailed debugging information.

Levels `1-3` are always sent to the Apache error log. Therefore you can always use level `0` as the default logging level in production. Level `5` is useful when debugging. It is not advisable to use higher logging levels in production as excessive logging can slow down server significantly.

SecDefaultAction

Defines the default action to take on a rule match. The default value is:

```
SecDefaultAction log,auditlog,deny,status:403,phase:2,\
    t:lowercase,t:replaceNulls,t:compressWhitespace
```

SecGuardianLog

Integration hook for `httpd-guardian` (see http://www.apachesecurity.net/tools/). For example:

```
SecGuardianLog |/path/to/httpd-guardian
```

SecRequestBodyAccess

Configures whether request bodies will be buffered and processed by ModSecurity by default. Possible values are:

- `On` - access request bodies.
- `Off` - do not attempt to access request bodies.

SecRequestBodyLimit

Configures the maximum request body size ModSecurity will accept for buffering. Anything over this limit will be rejected with status code 413 Request Entity Too Large.

SecRequestBodyInMemoryLimit

Configures the maximum request body size ModSecurity will store in memory. By default the limit is 128 KB:

```
# Store up to 128 KB in memory
SecRequestBodyInMemoryLimit 131072
```

SecResponseBodyLimit

Configures the maximum response body size that will be accepted for buffering. Anything over this limit will be rejected with status code 500 Internal Server Error. This setting will not affect the responses with MIME types that are not marked for buffering. By default this limit is configured to 512 KB:

```
# Buffer response bodies of up to 512 KB in length
SecResponseBodyLimit 524288
```

There is a hard limit of 1 GB.

SecResponseBodyMimeType

Configures which `MIME` types are to be considered for response body buffering. The default value is `text/plain text/html`:

```
SecResponseBodyMimeType text/plain text/html
```

Multiple `SecResponseBodyMimeType` directives can be used to add `MIME` types.

SecResponseBodyMimeTypesClear

Clears the list of `MIME` types considered for response body buffering, allowing you to start populating the list from scratch.

SecResponseBodyAccess

Configures whether response bodies are to be buffer and analysed or not. Possible values are:

- `On` - access response bodies (but only if the MIME type matches, see above).
- `Off` - do not attempt to access response bodies.

SecRule

`SecRule` is the main ModSecurity directive. It is used to analyse data and perform actions based on the results. In general, the format of this rule is as follows:

```
SecRule VARIABLES OPERATOR [ACTIONS]
```

The second part, `OPERATOR`, specifies how they are going to be checked. The third (optional) part, `ACTIONS`, specifies what to do whenever the operator used performs a successful match against a variable.

Variables in rules

The first part, `VARIABLES`, specifies which variables are to be checked. For example, the following rule will reject a transaction that has the word *dirty* in the URI:

```
SecRule REQUEST_URI dirty
```

Each rule can specify one or more variables:

```
SecRule REQUEST_URI|QUERY_STRING dirty
```

So far we have used only simple variables in our rules. Some variables are actually collections, which are expanded into more variables at runtime. The following example will examine all request arguments:

```
SecRule ARGS dirty
```

Sometimes, however, you will want to look only at parts of a collection. This can be achieved with the help of the *selection operator* (colon). The following example will only look at the arguments named `p` (do note that, in general, requests can contain multiple arguments with the same name):

```
SecRule ARGS:p dirty
```

It is also possible to specify exclusions. The following will examine all request arguments for the word *dirty*, except the ones named `z` (again, there can be zero or more arguments named `z`):

```
SecRule ARGS|!ARGS:z dirty
```

There is a special operator that allows you to count how many variables there are in a collection. The following rule will trigger if there is more than zero arguments in the request (ignore the second parameter for the time being):

```
SecRule &ARGS !^0$
```

And sometimes you need to look at an array of parameters, each with a slightly different name. In this case you can specify a regular expression in the selection operator itself. The following rule will look into all arguments whose names begin with `id_`:

```
SecRule ARGS:/^id_/ dirty
```

There is a third format supported by the selection operator - XPath expression. XPath expressions can only used against the special variable XML, which is available only of the request body was processed as XML.

```
SecRule XML:/xPath/Expression dirty
```

Note

As you have just seen, not all collections support all selection operator format types. You should refer to the documentation of each collection to determine what is and isn't supported.

Operators in rules

In the simplest possible case you will use a regular expression pattern as the second rule parameter. This is what we've done in the examples above. If you do this ModSecurity assumes you want to use the `rx` operator. You can explicitly specify the operator you want to use by using `@` as the first character in the second rule parameter:

```
SecRule REQUEST_URI "@rx dirty"
```

Note how we had to use double quotes to delimit the second rule parameter. This is because the second parameter now has a whitespace in it. Any number of whitespace characters can follow the name of the operator. If there are any non-whitespace characters there, they will all be treated as a special parameter to the operator. In the case of the regular expression operator the special parameter is the pattern that will be used for comparison.

The @ can be the second character if you are using negation to negate the result returned by the operator:

```
SecRule &ARGS "!@rx ^0$"
```

Actions in rules

The third parameter, `ACTIONS`, can be omitted only because there is a helper feature that specifies the default action list. If the parameter isn't omitted the actions specified in the parameter will be merged with the default action list to create the actual list of actions that will be processed on a rule match.

SecRuleInheritance

Configures whether the current context will inherit rules from the parent context (configuration options are inherited in most cases - you should look up the documentation for every directive to determine if it is inherited or not). Possible values are:

- `On` - inherit rules from the parent context.
- `Off` - do not inherit rules from the parent context.

Note

Resource-specific contexts (e.g. `Location`, `Directory`, etc) cannot override *phase 1* rules configured in the main server or in the virtual server. This is because phase 1 is run early in the request processing process, before Apache maps request to resource. Virtual host context can override phase 1 rules configured in the main server.

SecRuleEngine

Configures whether or not the ModSecurity Rule Engine will process transactions or not. Possible values are:

- `On` - process rules.
- `Off` - do not process rules.
- `DetectionOnly` - process rules but never intercept transactions, even when rules are configured to do so.

SecRuleRemoveById

Removes matching rules from the parent contexts. This directive supports multiple parameters, where each parameter can either be a rule ID, or a range. Parameters that contain spaces must be delimited using double quotes.

```
SecRuleRemoveById 1 2 5 10-20 "400 - 556" 673
```

SecRuleRemoveByMsg

Removes matching rules from the parent contexts. This directive supports multiple parameters. Each parameter is a regular expression that will be applied to the message (specified using the `msg` action).

SecServerSignature

Instructs ModSecurity to change web server signature.

```
SecServerSignature MyServer/1.0
```

SecTmpDir

Configures the directory where temporary files will be created.

SecUploadDir

Configures the directory where intercepted files will be stored. This directory must be on the same filesystem as the temporary directory defined with `SecTmpDir`.

SecUploadKeepFiles

Configures whether or not the intercepted files will be kept after transaction is processed. Possible values are:

- `On` -
- `Off` -
- `RelevantOnly` -

This directive requires the storage directory to be defined (using `SecUploadDir`).

SecWebAppId

Creates a partition on the server that belongs to one web application. Partitions are used to avoid collisions between session IDs and user IDs. This directive must be used if there are multiple applications deployed on the same server. If it isn't a collision between session IDs might occur. The default value is `default`.

```
SecWebAppId "Intranet"
```

Processing Phases

ModSecurity 2.x allows rules to be placed in one of the following five phases:

1. Request headers
2. Request body
3. Response headers
4. Response body
5. Logging

In order to select the phase a rule executes during, use the phase action either directly in the rule or in using the SecDefaultActions directive:

```
SecRule HTTP_Host "!^$" "deny,phase:1"
SecDefaultAction "log,pass,phase:2"
```

Phase Request Headers

Rules in this phase immediately after Apache completes reading the request headers. At this point the request body has not been read yet, meaning not all request arguments are available. Rules should be placed in this phase if you need to have them run early (before Apache does something with the request), to do something before the request body has been read, determine whether or not the request body should be buffered, or decide how you want the request body to be processed (e.g. whether to parse it as XML or not).

Phase Request Body

This is the general-purpose input analysis phase. Most of the application-oriented rules should go here. In this phase you are guaranteed to have received the request argument (provided

Phase Response Headers

This phase takes place just before response headers are sent back to the client. Run here if you want to observe the response before that happens, and if you want to use the response headers to determine if you want to buffer the response body.

Phase Response Body

This is the general-purpose output analysis phase. At this point you can run rules against the response body (provided it was buffered, of course).

Phase Logging

This phase is run just before logging takes place. The rules placed into this phase can only affect how the

logging is performed.

Variables

The following variables are supported in ModSecurity 2.x:

- ARGS - can be used on its own (means all arguments), with a static parameter (matches arguments with that name), or with a regular expression (matches all arguments with name that matches the regular expression). Note: ARGS:p will not result in any invocations against the operator if argument p does not exist.
- ARGS_COMBINED_SIZE -
- ARGS_NAMES -
- REQBODY_PROCESSOR - Built-in processors are URLENCODED, MULTIPART, and XML.
- REQBODY_ERROR - 0 or 1. If you want to stop processing on an error you must have an explicit rule in phase 2 to do so.
- REQBODY_ERROR_MSG - empty, or contains the error message from the processor.
- XML - can be used standalone (as a target for validateDTD and validateSchema) or with an XPath expression parameter (which makes it a valid target for any function that accepts plain text).
- WEBSERVER_ERROR_LOG - contains zero or more error messages produced by the web server.
- TX - Collection. This is where the transaction variables live.
- FILES - Collection. Contains a collection of original file names (as they were called on the remote user's file system).
- FILES_TMPNAMES - Collection. Contains a collection of temporary files' names on the disk. Useful when used together with @inspectFile (Note: only available if files were extracted from the request body.).
- FILES_NAMES - Collection w/o parameter. Contains a list of form fields that were used for file upload.
- FILES_SIZES - Collection. Contains a list of file sizes.
- FILES_COMBINED_SIZE - Single value. Total size of the uploaded files.
- ENV - Collection, requires a single parameter.
- REMOTE_HOST -
- REMOTE_ADDR -
- REMOTE_PORT -
- REMOTE_USER -
- PATH_INFO -
- QUERY_STRING -
- AUTH_TYPE -
- SERVER_NAME -
- SERVER_ADDR -
- SERVER_PORT -

- TIME_YEAR -
- TIME_EPOCH - time in seconds since 1970.
- TIME_MON -
- TIME_DAY -
- TIME_HOUR -
- TIME_MIN -
- TIME_SEC -
- TIME_WDAY -
- TIME -
- REQUEST_URI - (e.g. /index.php?p=X). This variable will never contain a domain name, even if it was provided on the request line. Warning: not urlDecoded.
- REQUEST_URI_RAW - same as above but will contain the domain name if it was provided on the request line (e.g. http://www.example.com/index.php?p=X). Warning: not urlDecoded.
- REQUEST_LINE -
- REQUEST_METHOD -
- REQUEST_PROTOCOL -
- REQUEST_FILENAME - relative REQUEST_URI minus the QUERY_STRING part (e.g. /index.php). Warning: not urlDecoded.
- REQUEST_BASENAME - just the filename part of REQUEST_FILENAME (e.g. index.php). Warning: not urlDecoded.
- SCRIPT_FILENAME -
- SCRIPT_BASENAME -
- SCRIPT_UID -
- SCRIPT_GID -
- SCRIPT_USERNAME -
- SCRIPT_GROUPNAME -
- SCRIPT_MODE -
- ENV -
- REQUEST_HEADERS -
- REQUEST_HEADERS_NAMES -
- REQUEST_COOKIES -
- REQUEST_COOKIES_NAMES -
- REQUEST_BODY -
- RESPONSE_LINE -
- RESPONSE_STATUS -
- RESPONSE_PROTOCOL -
- RESPONSE_HEADERS -

- `RESPONSE_HEADERS_NAMES` -
- `RESPONSE_BODY` -
- Special prefix `HTTP_` followed by a header name can be used to access any request header.
- `RULE` - Gives access to the `id`, `rev`, `severity`, and `msg` fields of the rule that triggered the action. Only available for expansion in action strings (e.g. `setvar:tx.varname=%{rule.id}`)
- `SESSION` - collection, available only after `setsid` is executed.
- `WEBAPPID` - the value set with `SecWebAppId`.
- `SESSIONID` - the value set with `setsid`.
- `USERID` - the value set with `setuid`.

Transformation functions

Transformation functions are used to transform a variable before testing it in a rule. The following rule will ensure that an attacker does not use mixed case in order to evade the ModSecurity rule:

```
SecRule ARG:p "xp_cmdshell" "t:lowercase"
```

multipe tranformation actions can be used in the same rule, for example the following rule also ensures that an attacker does not use URL encodign (%xx encoding) for evasion. Not the order of the transformation functions, which ensures that a URL encoded letter is first decoded and than translated to lower case.

```
SecRule ARG:p "xp_cmdshell" "t:urlDecode,t:lowercase"
```

One can use the SetDefaultAction command to ensure the translation occurs for every rule until the next. Note that translation actions are additive, so if a rule explicitly list actions, the translation actions set by SetDefaultAction are still performed.

```
SecDefaultAction t:urlDecode,t:lowercase
```

The following transformation functions are supported:

1. `lowercase` (enabled by default) - converts all charactes to lowercase using the current C locale.
2. `replaceNulls` (enabled by default) - replaces NULL bytes in input with spaces (32).
3. `removeNulls` - removes NULL bytes from input.
4. `compressWhitespace` (enabled by default) - converts whitespace characters (32, \f, \t, \n, \r, \v, 160) to spaces (32) and then compresses multiple space characters into only one.
5. `removeWhitespace` - removes all whitespace characters.
6. `replaceComments` - replaces each occurence of a C-style comments (`/* ... */`) with a single space (multiple consecutive occurences of a space will not be compressed). Unterminated comments will too be replaced with a space. However, a standalone termination of a comment (`*/`) will not be acted upon.
7. `urlDecode` - decodes an URL-encoded input string. Invalid encodings (i.e. the ones that use non-hexadecimal characters, or the ones that are at the end of string and have one or two characters missing) will not be converted. If you want to detect invalid encodings use the `@validateUrlEncoding` operator. The transformational function should not be used against variables that have already been URL-decoded unless it is your intention to perform URL decoding twice!
8. `urlEncode` - encodes input using URL encoding.
9. `urlDecodeUni` - In addition to decoding %xx like `urlDecode, urlDecodeUni also` decodes `%uXXXX` encoding (only the lower byte will be used, the higher byte will be discarded).

10. `base64Encode` - encodes input string using base64 encoding.
11. `base64Decode` - decoes a base64-encoded string.
12. `md5` - calculates an MD5 hash from input.
13. `sha1` - calculates a SHA1 hash from input.
14. `hexDecode` - decodes a hex-encoded string.
15. `hexEncode` - encodes input as hex-encoded string.
16. `htmlEntityDecode` - decodes HTML entities present in input. The following variants are supported:
    - `&#xHH` and `&#xHH;` (where H is any hexadecimal number)
    - `&#DDD` and `&#DDD;` (where D is any decimal number)
    - `&quot` and `&quot;`
    - `&nbs`p and ` `
    - `&lt` and `<`
    - `&gt` and `>`
17. `escapeSeqDecode` - decode ANSI C escape sequences: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\\`, `\?`, `\'`, `\"`, `\xHH` (hexadecimal), `\0OOO` (octal). Invalid encodings are left in the output.
18. `normalisePath` - will remove multiple slashes, self-references and directory back-references (except when they are at the beginning of the path).
19. `normalisePathWin` - as above, but will first convert backslash characters to forward slashes.
20. `none` - this not an actual transformation function but an instruction to ModSecurity to remove all transformation functions associated with the current rule and start from scratch.

Actions

Each action belongs to one of five groups:

1. *Disruptive actions*; can only appear in the first rule in a chain.
2. *Non-disruptive actions*; can appear anywhere.
3. *Flow actions*; can appear only in the first rule in a chain.
4. *Meta-data actions* (`id`, `rev`, `severity`, `msg`); can only appear in the first rule in a chain.
5. *Data actions* - can appear anywhere; these actions are completely passive and only serve to carry data used by other actions.

allow

Stops processing on a successful match and allows transaction to proceed.

auditlog

Marks the transaction for logging in the audit log.

capture

When used together with the regular expression operator capture action will create copies of regular expression captures and place them into the transaction variable collection. Up to ten captures will be copied on a successful pattern match, each with a name consisting of a digit from 0 to 9.

chain

Chains the rule where the action is placed with the rule that immediately follows it. The result is called a *rule chain*.

```
# Refuse to accept POST requests that do
# not specify request body length
SecRule REQUEST_METHOD ^POST$ chain
SecRule REQUEST_HEADER:Content-Length ^$
```

ctl

The ctl action allows configuration options to be updated for the transaction. The following configuration options are supported:

1. `auditEngine` -
2. `auditLogParts` -
3. `debugLogLevel` -
4. `requestBodyAccess` -

5. `requestBodyLimit` -
6. `requestBodyProcessor` -
7. `responseBodyAccess` -
8. `responseBodyLimit` -
9. `ruleEngine` -

With the exception of `requestBodyProcessor`, each configuration option corresponds to one configuration directive and the usage is identical.

The requestBodyProcessor option allows you to configure the request body processor. By default ModSecurity will use the `URLENCODED` and `MULTIPART` processors to process an `application/x-www-form-urlencoded` and a `multipart/form-data` body, respectively. A third processor, XML, is also supported, but it is never used implicitly. Instead you must tell ModSecurity to use it by placing a few rules in the `REQUEST_HEADERS` processing phase.

```
# Parse requests with Content-Type "text/xml" as XML
SecRule REQUEST_CONTENT_TYPE ^text/xml nolog,pass,ctl:requestBodyProcessor=XML
```

After the request body was processed as XML you will be able to use the XML-related features to inspect it.

Note

Request body processors will not interrupt a transaction if an error occurs during parsing. Instead they will set variables `REQBODY_PROCESSOR_ERROR` and `REQBODY_PROCESSOR_ERROR_MSG`. These variables should be inspected in the `REQUEST_BODY` phase and an appropriate action taken.

---

deny

Stops rule processing and intercepts transaction.

deprecatevar

Decrement counter based on its age. The following example will decrement the counter by 60 every 300 seconds.

```
deprecatevar:session.score=60/300
```

Counter values are always positive, meaning the value will never go below zero.

drop

Note: causes error message to appear in the log "(9)Bad file descriptor: core_output_filter: writing data to the network"

exec

Executes an external script/binary supplied as parameter.

expirevar

Configures collection variable to expire after the given time in seconds.

```
expirevar:session.suspicious=3600
```

id

Assigns a unique ID to the rule or chain.

initcol

Initialises a named persistent collection, either by loading data from storage or by creating a new collection in memory. The following example initiates IP address tracking.

```
initcol:ip=%{REMOTE_ADDR}
```

Every collection contains several built-in variables that are read-only:

1. `CREATE_TIME` -
2. `KEY` -
3. `LAST_UPDATE_TIME` -
4. `TIMEOUT` -
5. `UPDATE_COUNTER` -
6. `UPDATE_RATE` - collection updates per minute.

Collections are loaded into memory when the initcol action is encountered. The collection in storage will be updated (and the appropriate counters increased) *only* if it was changed during transaction processing.

Note

To create a collection to hold session variables (`SESSION`) use action `setsid`. To create a collection to hold user variables (`USER`) use action `setuid`.

Note

At this time it is only possible to have three collections: `IP`, `SESSION`, and `USER`.

---

log

Indicates that a successful match of the rule needs to be logged.

msg

Assigns a custom message to the rule or chain.

multiMatch

If enabled ModSecurity will perform multiple operator invocations for every target, before and after every anti-evasion transformation is performed.

noauditlog

Indicates that a successful match of the rule should not be used as criteria whether the transaction should be logged to the audit log.

nolog

Prevents rule matches from appearing in the log.

pass

Continues processing with the next rule in spite of a successful match. Transaction will not be interrupted but it will be logged (unless logging has been suppressed).

pause

Pauses transaction processing for the specified number of milliseconds.

phase

Places the rule (or the rule chain) into one of five available processing phases.

proxy

Intercepts transaction by forwarding request to another web server using the proxy backend.

redirect

Intercepts transaction by issuing a redirect to the given location. If the `status` action is present and its value is acceptable (301, 302, 303, or 307) it will be used for the redirection. Otherwise status code 302 will be used.

rev

Specifies rule revision. This action is used in combination with the `id` action to allow the same rule ID to be used after changes take place but to still provide some indication the rule changed.

sanitiseArg

Sanitises (replaces each byte with an asterisk) a named request argument prior to audit logging.

```
sanitiseArg:password
```

sanitiseMatched

Sanitises the variable (request argument, request header, or response header) that caused a rule match. This action can be used to sanitise arbitrary transaction elements when they match a condition. For example, the example below will sanitise any argument that contains the word *password* in the name.

```
SecRule ARGS_NAMES password nolog,pass,sanitiseMatched
```

sanitiseRequestHeader

Sanitises a named request header.

```
sanitiseRequestHeader:Authorization
```

sanitiseResponseHeader

Sanitises a names response header.

severity

Assigns severity to the rule it is placed with.

setuid

Special-purpose action that initialises the `USER` collection. After initialisation takes place the variable `USERID` will be available for use in the subsequent rules.

setsid

Special-purpose action that initialises the `SESSION` collection. On first invocation of this action the collection will be empty (not taking the pre-defined variables into account - see `initcol` for more information). On subsequent invocations the contents of the collection (session, in this case) will be retrieved from storage. After initialisation takes place the variable `SESSIONID` will be available for use in the subsequent rules.

```
# Initialise session variables using the session cookie value
SecRule REQUEST_COOKIES:PHPSESSID !^$ chain,nolog,pass
SecAction setsid:%{REQUEST_COOKIES.PHPSESSID}
```

This action understands each application maintains its own set of sessions. It will utilise the current web application ID to create a session namespace.

setenv

Creates, removes, or updates an environment variable. This action can be used to establish communication with other Apache modules.

To create a new variable (if you omit the value `1` will be used):

```
setenv:name=value
```

To remove a variable:

```
setenv:!name
```

setvar

Creates, removes, or updates a variable in the specified collection.

To create a new variable:

```
setvar:tx.score=10
```

To remove a variable prefix the name with exclamation mark:

```
setvar:!tx.score
```

To increase or decrease variable value use `+` and `-` characters in front of a numerical value:

```
setvar:tx.score=+5
```

skip

Skips one or more rules (or chains) on successful match. This action can not be used to skip rules within one chain.

Accepts a single paramater denoting the number of rules (or chains) to skip.

```
skip:3
```

status

Specifies the response status code to use with actions `deny` and `redirect`.

t

This action can be used which transformation function should be used against the specified variables before they (or the results, rather) are run against the operator specified in the rule.

xmlns

This action should be used together with an XPath expression to register a namespace.

Operators

A number of operators can be used in rules, as documented below.

eq

Numerical comparison.

ge

Numerical comparison.

gt

Numerical comparison.

inspectFile

Executes the external script/binary given as parameter to the operator against every file extracted from the request.

```
SecRule FILES_TMPNAMES "@inspectFile /opt/apache/bin/inspect_script.pl"
```

le

Numerical comparison.

lt

Numerical comparison.

rbl

Look up the parameter in the RBL given as parameter. Parameter can be an IPv4 address, or a hostname.

```
SecRule REMOTE_ADDR "@rbl sc.surbl.org"
```

rx

Regular expression operator. Regular expressions are handled by the PCRE library (http://www.pcre.org). ModSecurity compiles its regular expressions with the following settings:

1. The entire input is treated as a single line, even when there are newline characters present.
2. All matches are case-sensitive. If you do not care about case sensitivity you either need to implement the `lowercase` transformational function, or use the per-pattern `(?s)` modificator,

as allowed by PCRE.

3. The `PCRE_DOTALL` flag is set during compilation, meaning a single dot will match any character, including the newlines.

validateByteRange

Validates the byte range used in the variable falls into the specified range:

```
SecRule ARG:text "@validateByteRange 10, 13, 32-126"
```

validateDTD

```
SecRule XML "@validateDTD /path/to/file.dtd"
```

This operator requires request body to be processed as XML.

validateSchema

```
SecRule XML "@validateSchema /path/to/file.xsd"
```

This operator requires request body to be processed as XML.

validateUrlEncoding

Verifies the encodings used in the variable (if any) are valid.

validateUtf8Encoding

Verifies the variable is a valid UTF-8 encoded string.