ModSecurity Reference Manual

Version 2.1.6 / (February 19, 2008)

Table of Contents

Introduction 7
- HTTP Traffic Logging 7
- Real-Time Monitoring and Attack Detection 7
- Attack Prevention and Just-in-time Patching 7
- Flexible Rule Engine 8
- Embedded-mode Deployment 8
- Network-based Deployment 8
- Licensing 8

ModSecurity Core Rules 10
- Overview 10
- Core Rules Structure 10
- Core Rules Content 10

Installation 11

Configuration Directives 13
- `SecAction` 13
- `SecArgumentSeparator` 13
- `SecAuditEngine` 14
- `SecAuditLog` 14
- `SecAuditLog2` 15
- `SecAuditLogParts` 15
- `SecAuditLogRelevantStatus` 16

SecAuditLogStorageDir 16
SecAuditLogType 17
SecChrootDir 17
SecCookieFormat 17
SecDataDir 18
SecDebugLog 18
SecDebugLogLevel 18
SecDefaultAction 19
SecGuardianLog 19
SecRequestBodyAccess 20
SecRequestBodyLimit 20
SecRequestBodyInMemoryLimit 20
SecResponseBodyLimit 21
SecResponseBodyMimeType 21
SecResponseBodyMimeTypesClear 21
SecResponseBodyAccess 22
SecRule 22
SecRuleInheritance 23
SecRuleEngine 24
SecRuleRemoveById 25
SecRuleRemoveByMsg 25
SecServerSignature 25
SecTmpDir 25
SecUploadDir 26
SecUploadFileMode 26
SecUploadKeepFiles 26
SecWebAppId 27
Processing Phases 28
Phase Request Headers 29
Phase Request Body 29
Phase Response Headers 29
Phase Response Body 29
Phase Logging 30
Variables 31
ARGS 31
ARGS_COMBINED_SIZE 31
ARGS_NAMES 32
AUTH_TYPE 32
ENV 32
FILES 32
FILES_COMBINED_SIZE 33
FILES_NAMES 33

FILES_SIZES 33
FILES_TMPNAMES 33
HTTP_ 33
MULTIPART_CRLF_LF_LINES 33
MULTIPART_STRICT_ERROR 34
MULTIPART_UNMATCHED_BOUNDARY 34
PATH_INFO 35
QUERY_STRING 35
REMOTE_ADDR 35
REMOTE_HOST 35
REMOTE_PORT 35
REMOTE_USER 36
REQBODY_PROCESSOR 36
REQBODY_PROCESSOR_ERROR 36
REQBODY_PROCESSOR_ERROR_MSG 36
REQUEST_BASENAME 36
REQUEST_BODY 37
REQUEST_COOKIES 37
REQUEST_COOKIES_NAMES 37
REQUEST_FILENAME 37
REQUEST_HEADERS 37
REQUEST_HEADERS_NAMES 38
REQUEST_LINE 38
REQUEST_METHOD 38
REQUEST_PROTOCOL 38
REQUEST_URI 39
REQUEST_URI_RAW 39
RESPONSE_BODY 39
RESPONSE_HEADERS 39
RESPONSE_HEADERS_NAMES 39
RESPONSE_PROTOCOL 40
RESPONSE_STATUS 40
RULE 40
SCRIPT_BASENAME 40
SCRIPT_FILENAME 40
SCRIPT_GID 41
SCRIPT_GROUPNAME 41
SCRIPT_MODE 41
SCRIPT_UID 41
SCRIPT_USERNAME 41
SERVER_ADDR 42
SERVER_NAME 42

SERVER_PORT 42
SESSION 42
SESSIONID 42
TIME 43
TIME_DAY 43
TIME_EPOCH 43
TIME_HOUR 43
TIME_MIN 43
TIME_MON 43
TIME_SEC 43
TIME_WDAY 44
TIME_YEAR 44
TX 44
USERID 44
WEBAPPID 44
WEBSERVER_ERROR_LOG 45
XML 45
Transformation functions 47
base64Decode 47
base64Encode 47
compressWhitespace 47
escapeSeqDecode 47
hexDecode 48
hexEncode 48
htmlEntityDecode 48
lowercase 48
md5 48
none 48
normalisePath 48
normalisePathWin 48
removeNulls 48
removeWhitespace 49
replaceComments 49
replaceNulls 49
urlDecode 49
urlDecodeUni 49
urlEncode 49
sha1 49
Actions 50
allow 50
auditlog 50
capture 50

chain 51
ctl 51
deny 52
deprecatevar 52
drop 52
exec 53
expirevar 53
id 54
initcol 54
log 55
msg 55
multiMatch 56
noauditlog 56
nolog 56
pass 57
pause 57
phase 57
proxy 57
redirect 58
rev 58
sanitiseArg 58
sanitiseMatched 59
sanitiseRequestHeader 59
sanitiseResponseHeader 59
severity 59
setuid 60
setsid 60
setenv 60
setvar 61
skip 61
status 62
t 62
xmlns 62
Operators 63
eq 63
ge 63
gt 63
inspectFile 63
le 63
lt 64
rbl 64
rx 64

validateByteRange 64
validateDTD 65
validateSchema 65
validateUrlEncoding 65
validateUtf8Encoding 66
Miscellaneous Topics 67
Impedance Mismatch 67

Introduction

ModSecurity™ is a web application firewall (WAF). With over 70% of all attacks now carried out over the web application level, organisations need every help they can get in making their systems secure. WAFs are deployed to establish an external security layer that increases security, detects, and prevents attacks before they reach web applications. It provides protection from a range of attacks against web applications and allows for HTTP traffic monitoring and real-time analysis with little or no changes to existing infrastructure.

HTTP Traffic Logging

Web servers are typically well-equipped to log traffic in a form useful for marketing analyses, but fall short when it comes to logging of traffic to web applications. In particular, most are not capable of logging the request bodies. Your adversaries know this, and that is why most attacks are now carried out via POST requests, rendering your systems blind. ModSecurity makes full HTTP transaction logging possible, allowing complete requests and responses to be logged. Its logging facilities also allow fine-grained decisions to be made about exactly what is logged and when, ensure only the relevant data is recorded.

Real-Time Monitoring and Attack Detection

In addition to providing logging facilities, ModSecurity can monitor the HTTP traffic in real time in order to detect attacks. In this case ModSecurity operates as a web intrusion detection tool, allowing you to react to suspicious events that take place at your web systems.

Attack Prevention and Just-in-time Patching

ModSecurity can also act immediately to prevent attacks from reaching your web applications. There are three commonly used approaches:

1. Negative security model. Negative security model monitors requests for anomalies, unusual behaviour, and common web application attacks. It keeps anomaly scores for each request, IP addresses, application sessions, and user accounts. Requests with high anomaly scores are either logged or rejected altogether.
2. Positive security model. When positive security model is deployed, only requests that are known to be valid are accepted, with everything else rejected. This approach works best with applications that are heavily used but rarely updated.
3. Known weaknesses and vulnerabilities. Its rule language makes ModSecurity an ideal external patching tool. External patching is all about reducing the window of opportunity. Time needed to patch application vulnerabilities often runs to weeks in many organisations. With ModSecurity, applications can be patched from the outside, without touching the application source code (and even without any access to it), making your systems secure until a proper patch is produced.

Flexible Rule Engine

A flexible rule engine sits in the heart of ModSecurity. It implements the ModSecurity Rule Language, which is a specialised programming language designed to work with HTTP transaction data. The ModSecurity Rule Language was designed to be easy to use, yet flexible: common operations are simple while complex operations are possible. Certified ModSecurity Rules, included with subscription to ModSecurity, contain a comprehensive set of rules that implement general-purpose hardening, common web application security issues. Heavily commented, these rules can be used as a learning tool.

Embedded-mode Deployment

ModSecurity is an embeddable web application firewall, which means it can be deployed as part of your existing web server infrastructure provided your web servers are Apache-based. This deployment method has certain advantages:

1. No changes to existing network. It only takes a few minutes to add ModSecurity to your existing web servers. And because it was designed to be completely passive by default, you are free to deploy it incrementally and only use the features you need. It is equally easy to remove or deactivate it should decide you don't want it any more.
2. No single point of failure. Unlike with network-based deployments, you will not be introducing a new point of failure to your system.
3. Implicit load balancing and scaling. Because it works embedded in web servers, ModSecurity will automatically take advantage of the additional load balancing and scalability features. You will not need to think of load balancing and scaling unless your existing system needs them.
4. Minimal overhead. Because it works from inside the web server process there is no overhead for network communication and minimal overhead in parsing and data exchange.
5. No problem with encrypted or compressed content. Many IDS systems have difficulties analysing SSL traffic. This is not a problem for ModSecurity because it is positioned to work when the traffic is decrypted and decompressed.

ModSecurity is known to work well on a wide range of operating systems. Our customers are successfully running it on Linux, Windows, Solaris, FreeBSD, OpenBSD, NetBSD, AIX, Mac OS X, and HP-UX.

Network-based Deployment

ModSecurity works equally well when deployed as part of an Apache-based reverse proxy server, and many of our customers choose to do so. In this scenario, one installation of ModSecurity can protect any number of web servers (even the non-Apache ones).

Licensing

ModSecurity is available under two licenses. Users can choose to use the software under the terms of the GNU General Public License version 2 (licence text is included with the distribution), as an Open Source / Free Software product. A range of commercial licenses is also available, together with a range of com-

mercial support contracts. For more information on commercial licensing please contact Breach Security.

Note

ModSecurity, mod_security, and ModSecurity Pro are trademarks or registered trademarks of Breach Security, Inc.

ModSecurity Core Rules

Overview

ModSecurity is a web application firewall engine that provides very little protection on its own. In order to become useful, ModSecurity must be configured with rules. In order to enable users to take full advantage of ModSecurity out of the box, Breach Security Inc. is providing a free certified rule set for ModSecurity 2.0. Unlike intrusion detection and prevention systems, which rely on signature specific to known vulnerabilities, the Core Rules provide generic protection from unknown vulnerabilities often found in web applications, which are in most cases custom coded. The Core Rules are heavily commented to allow it to be used as a step-by-step deployment guide for ModSecurity. The latest Core Rules can be found at the ModSecurity website - http://www.modsecurity.org/projects/rules/.

Core Rules Structure

If you expect a single pack of Apache configuration files, you are right, and wrong. A ModSecurity rule set includes information about different areas:

- The logic required to detect attacks.
- A policy setting the actions to perform if an attack is detected.
- Information regarding attacks.

In order to allow separate management of the different parts, the Core Rules are based on templates that are generated into a run-time rule set by inserting policy, patterns and event information. The Core Rules package includes these templates, the generation script (written in Perl) and data files required to generate a useful rule set. It also includes a bunch of pre-generated rule sets for different policies. The generation script also allows two optimizations:

- Optimal use of regular expressions. Since regular expressions are much more efficient if assembled into a single expression and optimized, the generation script takes the list of patterns that are required for a rule and optimize them into a most efficient regular expression.
- Removal of rules that are not utilized by a specific policy.

Core Rules Content

In order to provide generic web applications protection, the Core Rules use the following techniques:

- HTTP protection - detecting violations of the HTTP protocol and a locally defined usage policy.
- Common Web Attacks Protection - detecting common web application security attack.
- Automation detection - Detecting bots, crawlers, scanners and other surface malicious activity.
- Trojan Protection - Detecting access to Trojans horses.
- Error Hiding - Disguising error messages sent by the server.

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. 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 the Apache ServerRoot directory. You can check this by identifying the ServerRoot directive setting in your httpd.conf file. This is the path that was specified with the "--install-path=" configuration flag during compilation (for example, in Fedora Core4: `top_dir = /etc/httpd`).
6. Edit Makefile to configure the correct include path for libxml (for example: `INCLUDES=-I/usr/include/libxml2`)
7. Compile with `make`
8. Stop Apache
9. Install with `make install`
10. 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 or are compiling for a distribution, please read the following notes.

The ModSecurity Core rules may assume XML support is available (compiled with -DWITH_LIBXML2). You may have to manually remove any XML references in the Core rules if you choose not to include XML support. In future versions of ModSecurity XML support will be required. For these reasons, please consider XML support required.

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.

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 regular expression processing.

Configuration Directives

The following section outlines all of the ModSecurity directives. Most of the ModSecurity directives can be used inside the various Apache Scope Directives such as `VirtualHost`, `Location`, `Location-Match`, `Directory`, etc... There are others, however, that can only be used once in the main configuration file. This information is specified in the Scope sections below.

These rules, along with the Core rules files, should be contained is files outside of the httpd.conf file and called up with Apache "Include" directives. This allows for easier updating/migration of the rules. If you create your own custom rules that you would like to use with the Core rules, you should create a file called - `modsecurity_crs_15_customrules.conf` and place it in the same directory as the Core rules files. By using this file name, your custom rules will be called up after the standard ModSecurity Core rules configuration file but before the other Core rules. This allows your rules to be evaluate first which can be useful if you need to implement specific "allow" rules or to correct any false positives in the Core rules as they are applied to your site.

Note

It is highly encouraged that you do not edit the Core rules files themselves but rather place all changes (such as `SecRuleRemoveByID`, etc...) in your custom rules file. This will allow for easier upgrading as newer Core rules are released by Breach Security on the ModSecurity website.

SecAction

Description: 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`.

Syntax: `SecAction action1,action2,action2`

Example Usage: `SecAction nolog,redirect:http://www.hostname.com`

ProcessingPhase: Any

Scope: Any

Dependencies/Notes: None

SecAction is best used when you uncondiationally execute an action. This is explicit triggering whereas the normal Actions are conditional based on data inspection of the request/response. This is a useful directive when you want to run certian actions such as initcol to initialize collections.

SecArgumentSeparator

Description: 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 (`;`).

Syntax: `SecArgumentSeparator character`

Example Usage: `SecArgumentSeparator ;`

Processing Phase: Any

Scope: Main

Dependencies/Notes: None

This directive is needed if a backend web appliaction is using a non-standard argument separator. If this directive is not set properly for each web app, then ModSecurity will not be able to parse the arguements appropriately and the effectiveness of the rule matching will be significantly decreased.

SecAuditEngine

Description: Configures the audit logging engine.

Syntax: `SecAuditEngine On|Off|RelevantOnly`

Example Usage: `SecAuditEngine On`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: Can be set/changed with the "ctl" action for the current transaction.

Example: The following example shows the various audit directives used together.

```
SecAuditEngine RelevantOnly
SecAuditLog logs/audit/audit.log
SecAuditLogParts ABCFHZ
SecAuditLogType concurrent
SecAuditLogStorageDir logs/audit
SecAuditLogRelevantStatus ^[45]
```

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

Description: Defines the path to the main audit log file.

Syntax: `SecAuditLog /path/to/auditlog`

Example Usage: `SecAuditLog /usr/local/apache/logs/audit.log`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: 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..

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. If you are planning to use Concurrent audit logging and sending your audit log data off to a remote Con-

sole host, then you will need to use the modsec-auditlog-collector.pl script and use the following format:

```
SecAuditLog \
  "|/path/modsec-auditlog-collector.pl /path/SecAuditLogDataDir /path/SecAuditLog"
```

SecAuditLog2

Description: Defines the path to the secondary audit log index file when concurrent logging is enabled. See `SecAuditLog2` for more details.

Syntax: `SecAuditLog2 /path/to/auditlog2`

Example Usage: `SecAuditLog2 /usr/local/apache/logs/audit2.log`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: A main audit log must be defined via `SecAuditLog` before this directive may be used. Additionally, this log is only used for replicating the main audit log index file when concurrent audit logging is used. It will **not** be used for non-concurrent audit logging.

SecAuditLogParts

Description: Defines the path to the main audit log file.

Syntax: `SecAuditLogParts PARTS`

Example Usage: `SecAuditLogParts ABCFHZ`

Processing Phase: N/A

Scope: Any

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

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)

SecAuditLogRelevantStatus

Description: Configures which response status code is to be considered relevant for the purpose of audit logging.

Syntax: `SecAuditLogRelevantStatus REGEX`

Example Usage: `SecAuditLogRelevantStatus ^[45]`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: Must have the SecAuditEngine set to RelevantOnly. The parameter is a regular expression.

The main purpose of this directive is to allow you to configure audit logging for only transactions that generate the specified HTTP Response Status Code. This directive is often used to the decrease the total size of the audit log file. Keep in mind that if this parameter is used, then successful attacks that result in a 200 OK status code will not be logged.

SecAuditLogStorageDir

Description: Configures the storage directory where concurrent audit log entries are to be stored.

Syntax: `SecAuditLogStorageDir /path/to/storage/dir`

Example Usage: `SecAuditLogStorageDir /usr/local/apache/logs/audit`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: SecAuditLogType must be set to Concurrent. The directory must already be created before starting Apache and it must be writable by the web server user as new files are generated at runtime.

As with all logging mechanisms, ensure that you specify a file system location that has adequate disk space and is not on the root partition.

SecAuditLogType

Description: Configures the type of audit logging mechanism to be used.

Syntax: `SecAuditLogType Serial|Concurrent`

Example Usage: `SecAuditLogType Serial`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: Must specify SecAuditLogStorageDir if you use concurrent logging.

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 separate files, one for each transaction. Concurrent logging is the mode to use if you are going to send the audit log data off to a remote ModSecurity Console host.

SecChrootDir

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

Syntax: `SecChrootDir /path/to/chroot/dir`

Example Usage: `SecChrootDir /chroot`

Processing Phase: N/A

Scope: Main

Dependencies/Notes: The internal chroot functionality provided by ModSecurity works great for simple setups. One example of a simple setup is Apache serving static files only, or running scripts using modules. For more complex setups you should consider building a jail the old-fashioned way. The internal chroot feature should be treated as somewhat experimental. Due to the large number of default and third-party modules available for the Apache web server, it is not possible to verify the internal chroot works reliably with all of them. You are advised to think about your option and make your own decision. In particular, if you are using any of the modules that fork in the module initialisation phase (e.g. mod_fastcgi, mod_fcgid, mod_cgid), you are advised to examine each Apache process and observe its current working directory, process root, and the list of open files.

SecCookieFormat

Description: Selects the cookie format that will be used in the current configuration context.

Syntax: `SecCookieFormat 0|1`

Example Usage: `SecCookieFormat 0`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: None

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

Description: Path where persistent data (e.g. IP address data, session data, etc) is to be stored.

Syntax: `SecDataDir /path/to/dir`

Example Usage: `SecDataDir /usr/local/apache/logs/data`

Processing Phase: N/A

Scope: Main

Dependencies/Notes: This directive is needed when initcol, setsid an setuid are used. Must be writable by the web server user.

SecDebugLog

Description: Path to the ModSecurity debug log file.

Syntax: `SecDebugLog /path/to/modsec-debug.log`

Example Usage: `SecDebugLog /usr/local/apache/logs/modsec-debug.log`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: None

SecDebugLogLevel

Description: Configures the verboseness of the debug log data.

Syntax: `SecDebugLogLevel 0|1|2|3|4|5|6|7|8|9`

Example Usage: `SecDebugLogLevel 4`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: 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.

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.

SecDefaultAction

Description: Defines the default action to take on a rule match.

Syntax: `SecDefaultAction action1,action2,action3`

Example Usage: `SecDefaultAction log,auditlog,deny,status:403,phase:2,t:lowercase`

Processing Phase: Any

Scope: Any

Dependencies/Notes: Rules following a SecDefaultAction directive will inherit this setting unless a specific action is specified for an indivdual rule or until another SecDefaultAction is specified.

The default value is:

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

Note

SecDefaultAction must specify a disruptive action and a processing phase.

SecGuardianLog

Description: Configuration directive to use the httpd-guardian script to monitor for Denial of Service (DoS) attacks.

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

Example Usage: `SecGuardianLog |/usr/local/apache/bin/httpd-guardian`

Processing Phase: N/A

Scope: Main

Dependencies/Notes: By default httpd-guardian will defend against clients that send more 120 requests in a minute, or more than 360 requests in five minutes.

Since 1.9, ModSecurity supports a new directive, SecGuardianLog, that is designed to send all access data to another program using the piped logging feature. Since Apache is typically deployed in a multi-process fashion, making information sharing difficult, the idea is to deploy a single external process to observe all requests in a stateful manner, providing additional protection.

Development of a state of the art external protection tool will be a focus of subsequent ModSecurity releases. However, a fully functional tool is already available as part of the Apache httpd tools project [http://www.apachesecurity.net/tools/]. The tool is called httpd-guardian and can be used to defend against Denial of Service attacks. It uses the blacklist tool (from the same project) to interact with an iptables-based (Linux) or pf-based (*BSD) firewall, dynamically blacklisting the offending IP addresses. It can also interact with SnortSam (http://www.snortsam.net). Assuming httpd-guardian is already con-

figured (look into the source code for the detailed instructions) you only need to add one line to your Apache configuration to deploy it:

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

SecRequestBodyAccess

Description: Configures whether request bodies will be buffered and processed by ModSecurity by default.

Syntax: `SecRequestBodyAccess On|Off`

Example Usage: `SecRequestBodyAccess On`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: This directive is required if you plan to inspect POST_PAYLOADS of requests. This directive must be used along with the "phase:2" processing phase action and REQUEST_BODY variable/location. If any of these 3 parts are not configured, you will not be able to inspect the request bodies.

Possible values are:

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

SecRequestBodyLimit

Description: Configures the maximum request body size ModSecurity will accept for buffering.

Syntax: `SecRequestBodyLimit NUMBER_IN_BYTES`

Example Usage: `SecRequestBodyLimit 134217728`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: 131072 KB (134217728 bytes) is the default setting. Anything over this limit will be rejected with status code 413 Request Entity Too Large. There is a hard limit of 1 GB.

SecRequestBodyInMemoryLimit

Description: Configures the maximum request body size ModSecurity will store in memory.

Syntax: `SecRequestBodyInMemoryLimit NUMBER_IN_BYTES`

Example Usage: `SecRequestBodyInMemoryLimit 131072`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: None

By default the limit is 128 KB:

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

SecResponseBodyLimit

Description: Configures the maximum response body size that will be accepted for buffering.

Syntax: `SecResponseBodyLimit NUMBER_IN_BYTES`

Example Usage: `SecResponseBodyLimit 524228`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: 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. There is a hard limit of 1 GB.

By default this limit is configured to 512 KB:

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

SecResponseBodyMimeType

Description: Configures which `MIME` types are to be considered for response body buffering.

Syntax: `SecResponseBodyMimeType mime/type`

Example Usage: `SecResponseBodyMimeType text/plain text/html`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: Multiple `SecResponseBodyMimeType` directives can be used to add `MIME` types.

The default value is `text/plaintext/html`:

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

SecResponseBodyMimeTypesClear

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

Syntax: `SecResponseBodyMimeTypesClear`

Example Usage: `SecResponseBodyMimeTypesClear`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: None

SecResponseBodyAccess

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

Syntax: `SecResponseBodyAccess On|Off`

Example Usage: `SecResponseBodyAccess On`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: This directive is required if you plan to inspect html responses. This directive must be used along with the "phase:4" processing phase action and RESPONSE_BODY variable/location. If any of these 3 parts are not configured, you will not be able to inspect the response bodies.

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

Description: `SecRule` is the main ModSecurity directive. It is used to analyse data and perform actions based on the results.

Syntax: `SecRule VARIABLES OPERATOR [ACTIONS]`

Example Usage: `SecRule REQUEST_URI "attack"`

Processing Phase: Any

Scope: Any

Dependencies/Notes: None

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
```

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

Description: 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).

Syntax: `SecRuleInheritance On|Off`

Example Usage: `SecRuleInheritance Off`

Processing Phase: Any

Scope: Any

Dependencies/Notes: Resource-specific contexts (e.g. `Location`, `Directory`, etc) cannot override *phase1* 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.

Example: The following example shows where ModSecurity may be enabled in the main Apache configuration scope, however you might want to configure your VirtualHosts differently. In the first example, the first virtualhost is not inheriting the ModSecurity main config directives and in the second one it is.

```
SecRuleEnine On
SecDefaultAction log,pass,phase:2
...

<VirtualHost *:80>
ServerName app1.com
ServerAlias www.app1.com
SecRuleInheritance Off
SecDefaultAction log,deny,phase:1,redirect:http://www.site2.com
...
</VirtualHost>

<VirtualHost *:80>
ServerName app2.com
ServerAlias www.app2.com
SecRuleInheritance On SecRule ARGS "attack"
...
</VirtualHost>
```

Possible values are:

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

SecRuleEngine

Description: Configures the rules engine.

Syntax: `SecRuleEngine On|Off|DetectionOnly`

Example Usage: `SecRuleEngine On`

Processing Phase: Any

Scope: Any

Dependencies/Notes: Thisdirective can also be controled by the ctl action (ctl:ruleEngine=off) for per rule processing.

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

Description: Removes matching rules from the parent contexts.

Syntax: `SecRuleRemoveById RULEID`

Example Usage: `SecRuleRemoveByID 1 2 "9000-9010"`

Processing Phase: Any

Scope: Any

Dependencies/Notes: 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

Description: Removes matching rules from the parent contexts.

Syntax: `SecRuleRemoveByMsg REGEX`

Example Usage: `SecRuleRemoveByMsg "FAIL"`

Processing Phase: Any

Scope: Any

Dependencies/Notes: This directive supports multiple parameters. Each parameter is a regular expression that will be applied to the message (specified using the `msg` action).

SecServerSignature

Description: Instructs ModSecurity to change the data presented in the "Server:" response header token.

Syntax: `SecServerSignature "WEB SERVER SOFTWARE"`

Example Usage: `SecServerSignature "Netscape-Enterprise/6.0"`

Processing Phase: N/A

Scope: Main

Dependencies/Notes: In order for this directive to work, you must set the Apache ServerTokens directive to Full. ModSecurity will overwrite the server signature data held in this memory space with the data set in this directive. If ServerTokens is not set to Full, then the memory space is most likely not large enough to hold the new data we are looking to insert.

SecTmpDir

Description: Configures the directory where temporary files will be created.

Syntax: `SecTmpDir /path/to/dir`

Example Usage: `SecTmpDir /tmp`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: Needs to be writable by the Apache user process. This is the directory location where Apache will swap data to disk if it runs out of memory (more data than what was specified in the SecRequestBodyInMemoryLimit directive) during inspection.

SecUploadDir

Description: Configures the directory where intercepted files will be stored.

Syntax: `SecUploadDir /path/to/dir`

Example Usage: `SecUploadDir /tmp`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: This directory must be on the same filesystem as the temporary directory defined with `SecTmpDir`. This directive is used with `SecUploadKeepFiles`.

SecUploadFileMode

Description: Configures the mode (permissions) of any uploaded files using an octal number.

Syntax: `SecUploadFileMode octal_mode|"default"`

Example Usage: `SecUploadFileMode 0640`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: The mode is an octal number (as used in chmod). The default mode is for only the account writing the file to have read/write access (0600). Use this directive with caution to avoid exposing potentially sensitive data to unauthorized users. Using the value "default" will revert back to the default setting.

SecUploadKeepFiles

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

Syntax: `SecUploadKeepFiles On|Off|RelevantOnly`

Example Usage: `SecUploadKeepFiles On`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: This directive requires the storage directory to be defined (using `SecUploadDir`).

Possible values are:

- `On` - Keep uploaded files.
- `Off` - Do not keep uploaded files.
- `RelevantOnly` - This will keep only those files that belong to requests that are deemed relevant.

SecWebAppId

Description: Creates a partition on the server that belongs to one web application.

Syntax: `SecWebAppId "NAME"`

Example Usage: `SecWebAppId "WebApp1"`

Processing Phase: N/A

Scope: Any

Dependencies/Notes: 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 used, a collision between session IDs might occur. The default value is `default`. Example:

```
<VirtualHost *:80>
ServerName app1.com
ServerAlias www.app1.com
SecWebAppId "App1"
SecRule REQUEST_COOKIES:PHPSESSID !^$ chain,nolog,pass
SecAction setsid:%{REQUEST_COOKIES.PHPSESSID}
...
</VirtualHost>

<VirtualHost *:80>
ServerName app2.com
ServerAlias www.app2.com
SecWebAppId "App2"
SecRule REQUEST_COOKIES:PHPSESSID !^$ chain,nolog,pass
SecAction setsid:%{REQUEST_COOKIES.PHPSESSID}
...
</VirtualHost>
```

In the two examples configurations shown, SecWebAppId is being used in conjuction with the Apache VirtualHost directives. What this achieves is to create more unique collection names when being hosted on one server. Normally, when setsid is used, ModSecurity will create a collection with the name "SESSION" and it will hold the value specified. With using SecWebAppId as shown in the examples, however, the name of the collection would become "App1_SESSION" and "App2_SESSION".

SecWebAppId is relevant in two cases:

1. You are logging transactions/alerts to the ModSecurity Console and you want to use the web application ID to search only the transactions belonging to that application.
2. You are using the data persistence facility (collections SESSION and USER) and you need to avoid collisions between sessions and users belonging to different applications.

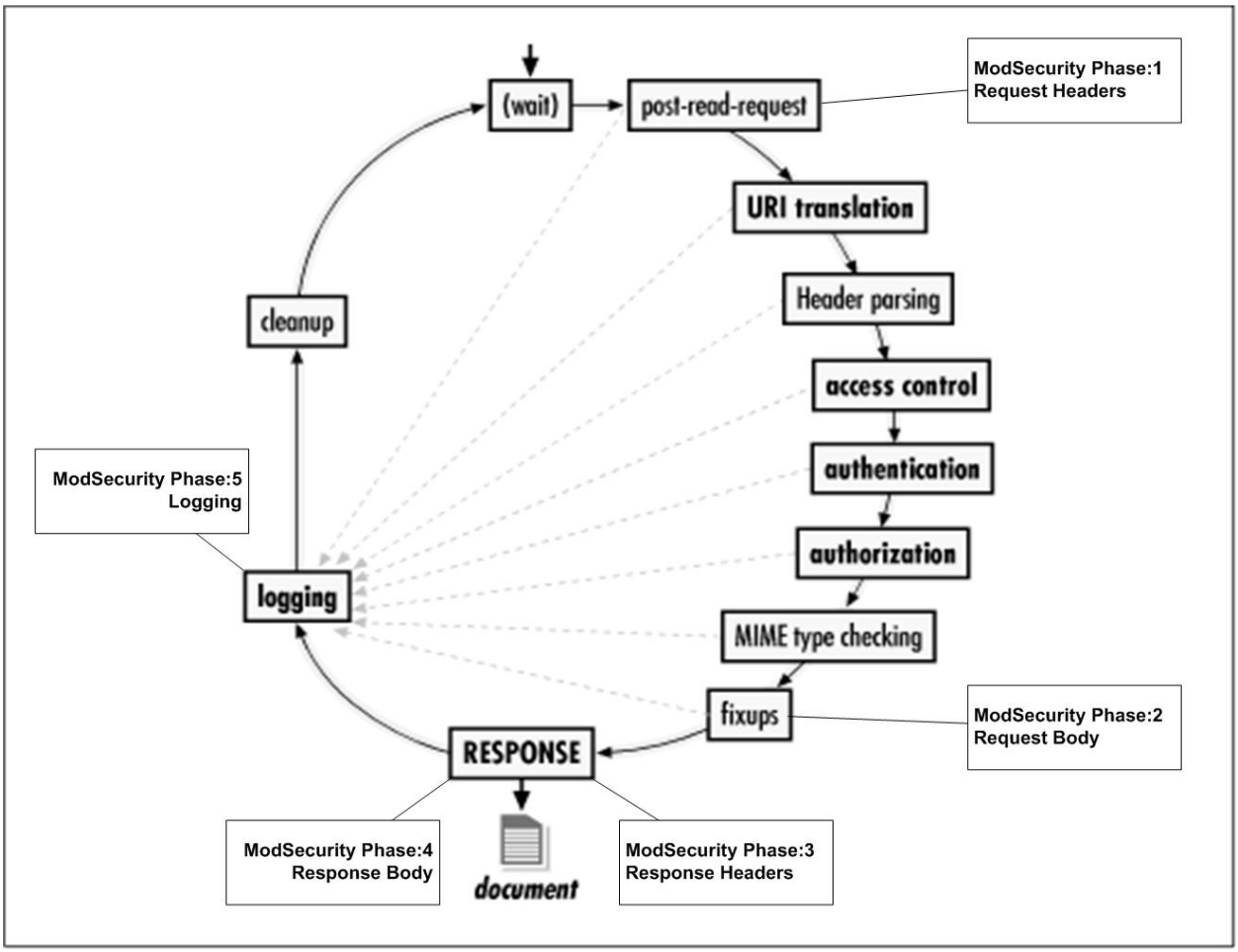
Processing Phases

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

ModSecurity Processing Phases Diagram

Below is a diagram of the standard Apache Request Cycle. In the diagram, the 5 ModSecurity processing phases are shown.



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

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

Note on Rule and Phases

Keep in mind that rules are executed according to phases, so even if two rules are adjacent in a configuration file, but are set to execute in different phases, they would not happen one after the other. The order of rules in the configuration file is important only within the rules of each phase. This is especially important when using the `skip` action.

Phase Request Headers

Rules in this phase are processed immediately after Apache completes reading the request headers (post-read-request phase). 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).

Note

Rules in this phase can not leverage Apache scope directives (Directory, Location, LocationMatch, etc...) as the post-read-request hook does not have this information yet. The exception here is the VirtualHost directive. If you want to use ModSecurity rules inside Apache locations, then they should run in Phase 2. Refer to the Apache Request Cycle/ModSecurity Processing Phases diagram.

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 the request body has been read). ModSecurity supports three encoding types for the request body phase:

- `application/x-www-form-urlencoded` - used to transfer form data
- `multipart/form-data` - used for file transfers
- `text/xml` - used for passing XML data

Other encodings are not used by most web applications.

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. Note that some response status codes (such as 404) are handled earlier in the request cycle by Apache and my not be able to be triggered as expected. Additionally, there are some response headers that are added by Apache at a later hook (such as Date, Server and Connection) that we would not be able to trigger on or sanitize. This should work appropirately in a proxy setup or within phase:5 (logging).

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). This is the phase where you would want to inspect the outbound html for information discloure, error messages or failed authentication text.

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. This phase can be used to inspect the error messages logged by Apache. You can not deny/block connections in this phase as it is too late. This phase also allows for inspection of other response headers that weren't available during phase:3 or phase:4.

Variables

The following variables are supported in ModSecurity 2.x:

ARGS

ARGS is a collection and can be used on its own (means all arguments including the POST Payload), 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. 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
```

Note

In ModSecurity 1.X, the ARGS variable stood for QUERY_STRING + POST_PAYLOAD, whereas now it expands to to individual variables.

ARGS_COMBINED_SIZE

This variable allows you to set more targeted evaluations on the total size of the Arguments as compared

with normal Apache LimitRequest directives. For example, you could create a rule to ensure that the total size of the argument data is below a certain threshold (to help prevent buffer overflow issues). Example: Block request if the size of the arguments is above 25 characters.

```
SecRule REQUEST_FILENAME "^/cgi-bin/login\.php$" "chain,log,deny,phase:2"
SecRule ARGS_COMBINED_SIZE "@gt 25"
```

ARGS_NAMES

Is a collection of the argument names. You can search for specific argument names that you want to block. In a positive policy scenario, you can also whitelist (using an inverted rule with the ! character) only authorized argument names. Example: This example rule will only allow 2 argument names - p and a. If any other argument names are injected, it will be blocked.

```
SecRule REQUEST_FILENAME "/index.php" "chain,log,deny,status:403,phase:2"
SecRule ARGS_NAMES "!^(p|a)$"
```

AUTH_TYPE

This variable holds the authentication method used to validate a user. Example:

```
SecRule AUTH_TYPE "basic" log,deny,status:403,phase:1,t:lowercase
```

Note

This data will not be available in a proxy-mode deployment as the authentication is not local. In a proxy-mode deployment, you would need to inpect the `REQUEST_HEADERS:Authorization` header.

ENV

Collection, requires a single parameter (after a colon character). The ENV variable is set with setenv and does not give access to the CGI environment variables. Example:

```
SecRule REQUEST_FILENAME "printenv" pass,setenv:tag=suspicious
SecRule ENV:tag "suspicious"
```

FILES

Collection. Contains a collection of original file names (as they were called on the remote user's file system). Note: only available if files were extracted from the request body. Example:

```
SecRule FILES "\.conf$" log,deny,status:403,phase:2
```

FILES_COMBINED_SIZE

Single value. Total size of the uploaded files. Note: only available if files were extracted from the request body. Example:

```
SecRule FILES_COMBINED_SIZE "@gt 1000" log,deny,status:403,phase:2
```

FILES_NAMES

Collection w/o parameter. Contains a list of form fields that were used for file upload. Note: only available if files were extracted from the request body. Example:

```
SecRule FILES_NAMES "^upfile$" log,deny,status:403,phase:2
```

FILES_SIZES

Collection. Contains a list of file sizes. Useful for implementing a size limitation on individual uploaded files. Note: only available if files were extracted from the request body. Example:

```
SecRule FILES_SIZES "@gt 100" log,deny,status:403,phase:2
```

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. Example:

```
SecRule FILES_TMPNAMES "@inspectFile /path/to/inspect_script.pl"
```

HTTP_

This variable is a special prefix that is followed by a header name and can be used to access any request header. Example:

```
SecRule HTTP_REFERER "www\.badsite\.com"
```

Note

This variable is for backward-compatibilty with ModSecurity 1.X rules. It has been superceded by the REQUEST_HEADERS variable (REQUEST_HEADERS:*Headername*)

MULTIPART_CRLF_LF_LINES

This flag variable will be set to `1` whenever a multipart request uses mixed line terminators. The `multipart/form-data` RFC requires `CRLF` sequence to be used to terminate lines. Since some client imple-

mentations use only LF to terminate lines you might want to allow them to proceed under certain circumstances (if you want to do this you will need to stop using MULTIPART_STRICT_ERROR and check each multipart flag variable individually, avoding MULTIPART_LF_LINE). However, mixing CRLF and LF line terminators is dangerous as it can allow for evasion. Therefore, in such cases, you will have to add a check for MULTIPART_CRLF_LF_LINES.

MULTIPART_STRICT_ERROR

MULTIPART_STRICT_ERROR will be set to 1 when any of the following variables is also set to 1: REQBODY_PROCESSOR_ERROR, MULTIPART_BOUNDARY_QUOTED, MULTIPART_BOUNDARY_WHITESPACE, MULTIPART_DATA_BEFORE, MULTIPART_DATA_AFTER, MULTIPART_HEADER_FOLDING, MULTIPART_LF_LINE, MULTIPART_SEMICOLON_MISSING. Each of these variables covers one unusual (although sometimes legal) aspect of the request body in multipart/form-data format. Your policies should *always* contain a rule to check either this variable (easier) or one or more individual variables (if you know exactly what you want to accomplish). Depending on the rate of false positives and your default policy you should decide whether to block or just warn when the rule is triggered.

The best way to use this variable is as in the example below:

```
SecRule MULTIPART_STRICT_ERROR "!@eq 0" \
"phase:2,t:none,log,deny,msg:'Multipart request body \
failed strict validation: \
PE %{REQBODY_PROCESSOR_ERROR}, \
BQ %{MULTIPART_BOUNDARY_QUOTED}, \
BW %{MULTIPART_BOUNDARY_WHITESPACE}, \
DB %{MULTIPART_DATA_BEFORE}, \
DA %{MULTIPART_DATA_AFTER}, \
HF %{MULTIPART_HEADER_FOLDING}, \
LF %{MULTIPART_LF_LINE}, \
SM %{MULTIPART_SEMICOLON_MISSING}'"
```

The multipart/form-data parser has been upgraded in ModSecurity v2.1.3 to actively look for signs of evasion. Many variables (as listed above) were added to expose various facts discovered during the parsing process. The MULTIPART_STRICT_ERROR variable is handy to check on all abnormalities at once. The individual variables allow detection to be fine-tuned according to your circumstances in order to reduce the number of false positives. Detailed analysis of various evasion techniques covered will be released as a separated document at a later date.

MULTIPART_UNMATCHED_BOUNDARY

Set to 1 when, during the parsing phase of a multipart/request-body, ModSecurity encounters what feels like a boundary but it is not. Such an event may occur when evasion of ModSecurity is attempted.

The best wasy to use this variable is as in the example below:

```
SecRule MULTIPART_UNMATCHED_BOUNDARY "!@eq 0" \
"phase:2,t:none,log,deny,msg:'Multipart parser detected a possible unmatched boundary.'"
```

Change the rule from blocking to logging-only if many false positives are encountered.

PATH_INFO

Besides passing query information to a script/handler, you can also pass additional data, known as extra path information, as part of the URL. Example:

```
SecRule PATH_INFO "^/(bin|etc|sbin|opt|usr)"
```

QUERY_STRING

This variable holds form data passed to the script/handler by appending data after a question mark. Warning: Not URL-decoded. Example:

```
SecRule QUERY_STRING "attack"
```

REMOTE_ADDR

This variable holds the IP address of the remote client. Example:

```
SecRule REMOTE_ADDR "^192\.168\.1\.101$"
```

REMOTE_HOST

If HostnameLookUps are set to On, then this variable will hold the DNS resolved remote host name. If it is set to Off, then it will hold the remote IP address. Possible uses for this variable would be to deny known bad client hosts or network blocks, or conversely, to allow in authorized hosts. Example:

```
SecRule REMOTE_HOST "\.evil\.network\org$"
```

REMOTE_PORT

This variable holds information on the source port that the client used when initiating the connection to our web server. Example: in this example, we are evaluating to see if the `REMOTE_PORT` is less than 1024, which would indicate that the user is a privileged user (root).

```
SecRule REMOTE_PORT "@lt 1024" phase:1,log,pass,setenv:remote_port=privileged
```

REMOTE_USER

This variable holds the username of the authenticated user. If there are no password (basic|digest) access controls in place, then this variable will be empty. Example:

```
SecRule REMOTE_USER "admin"
```

Note

This data will not be available in a proxy-mode deployment as the authentication is not local.

REQBODY_PROCESSOR

Built-in processors are `URLENCODED`, `MULTIPART`, and `XML`. Example:

```
SecRule REQBODY_PROCESSOR "^XML$ chain
SecRule XML "@validateDTD /opt/apache-frontend/conf/xml.dtd"
```

REQBODY_PROCESSOR_ERROR

Possible values are 0 (no error) or 1 (error). This variable will be set by request body processors (typically the `multipart/request-data` parser or the XML parser) when they fail to properly parse a request payload.

Example:

```
SecRule REQBODY_PROCESSOR_ERROR "@eq 1" deny,phase:2
```

Note

Your policies *must* have a rule to check REQBODY_PROCESSOR_ERROR at the beginning of phase 2. Failure to do so will leave the door open for impedance mismatch attacks. It is possible, for example, that a payload that cannot be parsed by ModSecurity can be successfully parsed by more tolerant parser operating in the application. If your policy dictates blocking then you should reject the request if error is detected. When operating in detection-only mode your rule should alert with high severity when request body processing fails.

REQBODY_PROCESSOR_ERROR_MSG

Empty, or contains the error message from the processor. Example:

```
SecRule REQBODY_PROCESSOR_ERROR_MSG "failed to parse" t:lowercase
```

REQUEST_BASENAME

This variable holds just the filename part of `REQUEST_FILENAME` (e.g. index.php). Warning: not url-Decoded. Example:

```
SecRule REQUEST_BASENAME "^login\.php$"
```

REQUEST_BODY

This variable holds the data in the request body (including POST_PAYLOAD data). REQUEST_BODY should be used if the original order of the arguements is important (ARGS should be used in all other cases). Example:

```
SecRule REQUEST_BODY "^username=\w{25,}\&password=\w{25,}\&Submit\=login$"
```

Note

This variable is only available if the content type is application/x-www-form-urlencoded.

REQUEST_COOKIES

This variable is a collection of all of the cookie data. Example: the following example is using the Ampersand special operator to count how many variables are in the collection. In this rule, it would trigger if the request does not include any Cookie headers.

```
SecRule &REQUEST_COOKIES "@eq 0"
```

REQUEST_COOKIES_NAMES

This variable is a collection of the cookie names in the request headers. Example: the following rule will trigger if the JSESSIONID cookie is not present.

```
SecRule &REQUEST_COOKIES_NAMES:JSESSIONID "@eq 0"
```

REQUEST_FILENAME

This variable holds the relative REQUEST_URI minus the QUERY_STRING part (e.g. /index.php). Example:

```
SecRule REQUEST_FILENAME "^/cgi-bin/login\.php$"
```

REQUEST_HEADERS

This variable can be used as either a collection of all of the Request Headers or can be used to specify indivudual headers (by using REQUEST_HEADERS*:Header-Name*). Example: the first example uses REQUEST_HEADERS as a collection and is applying the validateUrlEncoding operator against all headers.

```
SecRule REQUEST_HEADERS "@validateUrlEncoding"
```

Example: the second example is targeting only the Host header.

```
SecRule REQUEST_HEADERS:Host "^[\d\.]+$" \
    "deny,log,status:400,msg:'Host header is a numeric IP address'"
```

REQUEST_HEADERS_NAMES

This variable is a collection of the names of all of the Request Headers. Example:

```
SecRule REQUEST_HEADERS_NAMES "^x-forwarded-for" \
    "log,deny,status:403,t:lowercase,msg:'Proxy Server Used'"
```

REQUEST_LINE

This variable holds the complete request line sent to the server (including the REQUEST_METHOD and HTTP version data). Example: this example rule will trigger if the request method is something other than GET, HEAD, POST or if the HTTP is something other than HTTP/0.9, 1.0 or 1.1.

```
SecRule REQUEST_LINE "!(^((?:(?:pos|ge)t|head))|http/(0\.9|1\.0|1\.1)$)"
```

Note

Due to the default action transformation function lowercase, the regex strings should be in lowercase as well unless the t:none transformation function is specified for this particular rule.

REQUEST_METHOD

This variable holds the Request Method used by the client. Example: the following example will trigger if the Request Method is either CONNECT or TRACE.

```
SecRule REQUEST_METHOD "^((?:connect|trace))$"
```

Note

Due to the default action transformation function lowercase, the regex strings should be in lowercase as well unless the t:none transformation function is specified for this particular rule.

REQUEST_PROTOCOL

This variable holds the Request Protocol Version information. Example:

```
SecRule REQUEST_PROTOCOL "!^http/(0\.9|1\.0|1\.1)$"
```

Note

Due to the default action transformation function lowercase, the regex strings should be in lowercase as well unless the t:none transformation function is specified for this particular rule.

REQUEST_URI

This variable holds the full URL including the QUERY_STRING data (e.g. /index.php?p=X), however it will never contain a domain name, even if it was provided on the request line. Warning: not urlDecoded. It also does not include either the REQUEST_METHOD or the HTTP version info. Example:

```
SecRule REQUEST_URI "attack"
```

REQUEST_URI_RAW

Same as REQUEST_URI 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. Example:

```
SecRule REQUEST_URI_RAW "http:/"
```

RESPONSE_BODY

This variable holds the data for the response payload. Example:

```
SecRule RESPONSE_BODY "ODBC Error Code"
```

RESPONSE_HEADERS

This variable is similar to the REQUEST_HEADERS variable and can be used in the same manner. Example:

```
SecRule RESPONSE_HEADERS:X-Cache "MISS"
```

Note

This variable may not have access to some headers when running in embedded-mode. Headers such as Server, Date, Connection and Content-Type are added during a later Apache hook just prior to sending the data to the client. This data should be available, however, either during ModSecurity phase:5 (logging) or when running in proxy-mode.

RESPONSE_HEADERS_NAMES

This variable is a collection of the response header names. Example:

```
SecRule RESPONSE_HEADERS_NAMES "Set-Cookie"
```

Note

Same limitations as RESPONSE_HEADERS with regards to access to some headers in embedded-mode.

RESPONSE_PROTOCOL

This variable holds the HTTP Response Protocol information. Example:

```
SecRule RESPONSE_PROTOCOL "^HTTP\/0\.9"
```

RESPONSE_STATUS

This variable holds the HTTP Response Status Code generated by Apache. Example:

```
SecRule RESPONSE_STATUS "^[45]"
```

Note

This directive may not work as expected in embedded-mode as Apache handles many of the stock response codes (404, 401, etc...) earlier in Phase 2. This variable should work as expected in a proxy-mode deployment.

RULE

This variable provides 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}`). Example:

```
SecRule &REQUEST_HEADERS:Host "@eq 0" "log,deny,setvar:tx.varname=%{rule.id}"
```

SCRIPT_BASENAME

This variable holds just the local filename part of SCRIPT_FILENAME. Example:

```
SecRule SCRIPT_BASENAME "^login\.php$"
```

Note

This variable is not available in proxy mode.

SCRIPT_FILENAME

This variable holds the full path on the server to the requested script. (e.g. SCRIPT_NAME plus the server path). Example:

```
SecRule SCRIPT_FILENAME "^/usr/local/apache/cgi-bin/login\.php$"
```

Note

This variable is not available in proxy mode.

SCRIPT_GID

This variable holds the groupid (numerical value) of the group owner of the script. Example:

```
SecRule SCRIPT_GID "!^46$"
```

Note

This variable is not available in proxy mode.

SCRIPT_GROUPNAME

This variable holds the group name of the group owner of the script. Example:

```
SecRule SCRIPT_GROUPNAME "!^apache$"
```

Note

This variable is not available in proxy mode.

SCRIPT_MODE

This variable holds the script's permissions mode data (numerical - 1=execute, 2=write, 4=read and 7=read/write/execute). Example: will trigger if the script has the WRITE permissions set.

```
SecRule SCRIPT_MODE "^(2|3|6|7)$"
```

Note

This variable is not available in proxy mode.

SCRIPT_UID

This variable holds the userid (numerical value) of the owner of the script. Example: the example rule below will trigger if the UID is not 46 (the Apache user).

```
SecRule SCRIPT_UID "!^46$"
```

Note

This variable is not available in proxy mode.

SCRIPT_USERNAME

This variable holds the username of the owner of the script. Example:

```
SecRule SCRIPT_USERNAME "!^apache$"
```

Note

This variable is not available in proxy mode.

SERVER_ADDR

This variable contains the IP address of the server. Example:

```
SecRule SERVER_ADDR "^192\.168\.1\.100$"
```

SERVER_NAME

This variable contains the server's hostname or IP address. Example:

```
SecRule SERVER_NAME "hostname\.com$"
```

Note

This data is taken from the Host header submitted in the client request.

SERVER_PORT

This variable contains the local port that the web server is listening on. Example:

```
SecRule SERVER_PORT "^80$"
```

SESSION

This variable is a collection, available only after `setsid` is executed. Example: the following example shows how to initialize a SESSION collection with setsid, how to use setvar to increase the session.score values, how to set the session.blocked variable and finally how to deny the connection based on the session:blocked value.

```
SecRule REQUEST_COOKIES:PHPSESSID !^$ chain,nolog,pass
SecAction setsid:%{REQUEST_COOKIES.PHPSESSID}
SecRule REQUEST_URI "^/cgi-bin/finger$" "pass,log,setvar:session.score=+10"
SecRule SESSION:SCORE "@gt 50" "pass,log,setvar:session.blocked=1"
SecRule SESSION:BLOCKED "@eq 1" "log,deny,status:403"
```

SESSIONID

This variable is the value set with `setsid`. Example:

```
SecRule SESSIONID !^$ chain,nolog,pass
SecRule REQUEST_COOKIES:PHPSESSID !^$
SecAction setsid:%{REQUEST_COOKIES.PHPSESSID}
```

TIME

This variable holds a formatted string representing the time (hour:minute:second). Example:

```
SecRule TIME "^(([1](8|9))|([2](0|1|2|3))):\d{2}:\d{2}$"
```

TIME_DAY

This variable holds the current date (1-31). Example: this rule would trigger anytime between the 10th and 20th days of the month.

```
SecRule TIME_DAY "^(([1](0|1|2|3|4|5|6|7|8|9))|20)$"
```

TIME_EPOCH

This variable holds the time in seconds since 1970. Example:

```
SecRule TIME_EPOCH "@gt 1000"
```

TIME_HOUR

This variable holds the current hour (0-23). Example: this rule would trigger during "off hours".

```
SecRule TIME_HOUR "^(0|1|2|3|4|5|6|[1](8|9)|[2](0|1|2|3))$"
```

TIME_MIN

This variable holds the current minute (0-59). Example: this rule would trigger during the last half hour of every hour.

```
SecRule TIME_MIN "^(3|4|5)"
```

TIME_MON

This variable holds the current month (0-11). Example: this rule would match if the month was either November (10) or December (11).

```
SecRule TIME_MON "^1"
```

TIME_SEC

This variable holds the current second count (0-59). Example:

```
SecRule TIME_SEC "@gt 30"
```

TIME_WDAY

This variable holds the current weekday (0-6). Example: this rule would trigger only on week-ends (Saturday and Sunday).

```
SecRule TIME_WDAY "^(0|6)$"
```

TIME_YEAR

This variable holds the current four-digit year data. Example:

```
SecRule TIME_YEAR "^2006$"
```

TX

Transaction Collection. This is used to store pieces of data, create a transaction anomaly score, and so on. Transaction variables are set for 1 request/response cycle. The scoring and evaluation will not last past the current request/response process. Example: In this example, we are using setvar to increase the tx.score value by 5 points. We then have a follow-up run that will evaluate the transactional score this this request and then it will decided whether or not to allow/deny the request through.

```
SecRule WEBSERVER_ERROR_LOG "does not exist" "phase:5,pass,setvar:tx.score=+5"
SecRule TX:SCORE "@gt 20" deny,log
```

USERID

This variable is the value set with `setuid`. Example:

```
SecAction setuid:%{REMOTE_USER},nolog
SecRule USERID "Admin"
```

WEBAPPID

This variable is the value set with `SecWebAppId`. Example:

```
SecWebAppId "WebApp1"
SecRule WEBAPPID "WebApp1" "chain,log,deny,status:403"
SecRule REQUEST_HEADERS:Transfer-Encoding "!^$"
```

WEBSERVER_ERROR_LOG

Contains zero or more error messages produced by the web server. Access to this variable is in phase:5 (logging). Example:

```
SecRule WEBSERVER_ERROR_LOG "File does not exist" "phase:5,setvar:tx.score=+5"
```

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). Example using XPath:

```
SecDefaultAction log,deny,status:403,phase:2
SecRule REQUEST_HEADERS:Content-Type ^text/xml$ \
    phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML
SecRule REQBODY_PROCESSOR "!^XML$" skip:2
SecRule XML:/employees/employee/name/text() Fred
SecRule XML:/xq:employees/employee/name/text() Fred \
    xmlns:xq=http://www.example.com/employees
```

The first XPath expression does not use namespaces. It would match against payload such as this one:

```
<employees>
    <employee>
        <name>Fred Jones</name>
        <address location="home">
            <street>900 Aurora Ave.</street>
            <city>Seattle</city>
            <state>WA</state>
            <zip>98115</zip>
        </address>
        <address location="work">
            <street>2011 152nd Avenue NE</street>
            <city>Redmond</city>
            <state>WA</state>
            <zip>98052</zip>
        </address>
        <phone location="work">(425)555-5665</phone>
        <phone location="home">(206)555-5555</phone>
        <phone location="mobile">(206)555-4321</phone>
    </employee>
</employees>
```

The second XPath expression does use namespaces. It would match the following payload:

```
<xq:employees xmlns:xq="http://www.example.com/employees">
    <employee>
        <name>Fred Jones</name>
```

```
        <address location="home">
            <street>900 Aurora Ave.</street>
            <city>Seattle</city>
            <state>WA</state>
            <zip>98115</zip>
        </address>
        <address location="work">
            <street>2011 152nd Avenue NE</street>
            <city>Redmond</city>
            <state>WA</state>
            <zip>98052</zip>
        </address>
        <phone location="work">(425)555-5665</phone>
        <phone location="home">(206)555-5555</phone>
        <phone location="mobile">(206)555-4321</phone>
    </employee>
</xq:employees>
```

Note the different namespace used in the second example.

To learn more about XPath we suggest the following resources:

1. XPath Standard [http://www.w3.org/TR/xpath]
2. XPath Tutorial [http://www.zvon.org/xxl/XPathTutorial/General/examples.html]

Transformation functions

When ModSecurity receives request or response information, it makes a copy of this data and places it into memory. It is on this data in memory that transformation functions are applied. The raw request/response data is never altered. Transformation functions are used to transform a variable before testing it in a rule.

Note

The default transformation function setting is - lowercase, replaceNulls and compressWhitespace (in this order).

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"
```

multiple tranformation actions can be used in the same rule, for example the following rule also ensures that an attacker does not use URL encoding (%xx encoding) for evasion. Note 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:

base64Decode

This function decoes a base64-encoded string.

base64Encode

This function encodes input string using base64 encoding.

compressWhitespace

This function is enabled by default. It converts whitespace characters (32, \f, \t, \n, \r, \v, 160) to spaces (ASCII 32) and then compresses multiple space characters into only one.

escapeSeqDecode

This function decode ANSI C escape sequences: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\\`, `\?`, `\'`, `\"`, `\xHH` (hexadecimal), `\0OOO` (octal). Invalid encodings are left in the output.

hexDecode

This function decodes a hex-encoded string.

hexEncode

This function encodes input as hex-encoded string.

htmlEntityDecode

This function 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;`
- ` ` and ` `
- `&lt` and `<`
- `&gt` and `>`

lowercase

This function is enabled by default. It converts all charactes to lowercase using the current C locale.

md5

This function calculates an MD5 hash from input.

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.

normalisePath

This function will remove multiple slashes, self-references and directory back-references (except when they are at the beginning of the path).

normalisePathWin

Same as normalisePath, but will first convert backslash characters to forward slashes.

removeNulls

This function removes NULL bytes from input.

removeWhitespace

This function removes all whitespace characters.

replaceComments

This function 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 (ASCII 32). However, a standalone termination of a comment (`*/`) will not be acted upon.

replaceNulls

This function is enabled by default. It replaces NULL bytes in input with spaces (ASCII 32).

urlDecode

This function 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!

urlDecodeUni

In addition to decoding %xx like `urlDecode`, `urlDecodeUni` also decodes `%uXXXX` encoding. If the code is in the range of FF01-FF5E (the full width ASCII codes), then the higher byte is used to detect and adjust the lower byte. Otherwise, only the lower byte will be used and the higher byte zeroed.

urlEncode

This function encodes input using URL encoding.

sha1

This function calculates a SHA1 hash from input.

Actions

Each action belongs to one of five groups:

1. *Disruptive actions* - are those actions where ModSecurity will intercept the data. They 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`

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

Action Group: Disruptive

Example:

```
SecRule REMOTE_ADDR "^192\.168\.1\.100$" nolog,phase:1,allow
```

Note

The allow action only applies to the current processing phase. If your intent is to explicitly allow a request, then you should use the "ctl" action to turn the ruleEngine off - `ctl:ruleEngine=Off`.

`auditlog`

Description: Marks the transaction for logging in the audit log.

Action Group: Non-Disruptive

Example:

```
SecRule REMOTE_ADDR "^192\.168\.1\.100$" auditlog,phase:1,allow
```

Note

The auditlog action is now explicit if log is already specified.

`capture`

Description: 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.

Action Group: Non-Disruptive

Example:

```
SecRule REQUEST_BODY "^username=(\w{25,})" phase:2,capture,t:none,chain
SecRule TX:1 "(?:(?:a(dmin|nonymous)))"
```

Note

The 0 data captures the entire REGEX match and 1 captures the data in the first parantheses, etc...

chain

Description: Chains the rule where the action is placed with the rule that immediately follows it. The result is called a *rule chain*. Chained rules allow for more complex rule matches where you want to use a number of different VARIABLES to create a better rule and to help prevent false positives.

Action Group: Flow

Example:

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

Note

In programming language concepts, think of chained rules somewhat similar to AND conditional statements. The actions specified in the first portion of the chained rule will only be triggered if all of the variable checks return positive hits. If one aspect of the chained rule is negative, then the entire rule chain is negative. Also note that disruptive actions, execution phases, metadata actions (id, rev, msg) and skip actions can only be specified on by the chain starter rule.

ctl

Description: The ctl action allows configuration options to be updated for the transaction.

Action Group: Non-Disruptive

Example:

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

Note

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

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

Description: Stops rule processing and intercepts transaction.

Action Group: Disruptive

Example:

```
SecRule REQUEST_HEADERS:User-Agent "nikto" "log,deny,msg:'Nikto Scanners Identified'"
```

deprecatevar

Description: Decrement counter based on its age.

Action Group: Non-Disruptive

Example: The following example will decrement the counter by 60 every 300 seconds.

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

Note

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

drop

Description: Immediately initiate a "connection close" action to tear down the TCP connection by sending a FIN packet.

Action Group: Disruptive

Example: The following example initiates an IP collection for tracking Basic Authentication attempts. If

the client goes over the threshold of more than 25 attempts in 2 minutes, it will DROP subsequent connections.

```
SecAction initcol:ip=%{REMOTE_ADDR},nolog
SecRule ARGS:login "!^$" \
    nolog,phase:1,setvar:ip.auth_attempt=+1,deprecatevar:ip.auth_attempt=20/120
SecRule IP:AUTH_ATTEMPT "@gt 25" \
    log,drop,phase:1,msg:'Possible Brute Force Attack"
```

Note

This action is extremely useful when responding to both Brute Force and Denial of Service attacks in that, in both cases, you want to minimize both the network bandwidth and the data returned to the client. This action causes error message to appear in the log "(9)Bad file descriptor: core_output_filter: writing data to the network"

exec

Description: Executes an external script/binary supplied as parameter.

Action Group: Non-Disruptive

Example:

```
SecRule REQUEST_URI "^/cgi-bin/script\.pl" \
    "log,exec:/usr/local/apache/bin/test.sh,phase:1"
```

Note

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. The script you execute must write something (anything) to stdout. If it doesn't ModSecurity will assume execution didn't work.

expirevar

Description: Configurescollection variable to expire after the given time in seconds.

Action Group: Non-Disruptive

Example:

```
SecRule REQUEST_COOKIES:JSESSIONID "!^$" nolog,phase:1,pass,chain
SecAction setsid:%{REQUEST_COOKIES:JSESSIONID}
SecRule REQUEST_URI "^/cgi-bin/script\.pl" \
    "log,allow,setvar:session.suspicious=1,expirevar:session.suspicious=3600,phase:1"
```

Note

You should use expirevar actions at the same time that you use setvar actions in order to keep the indended expiration time. If they are used on their own (perhaps in a SecAction directive) the expire time could get re-set. When variables are removed from collections, and there are no other changes, collections are not written to disk at the end of request. This is because the variables can always be expired again when the collection is read again on a subsequent request.

id

Description: Assigns a unique ID to the rule or chain.

Action Group: Metadata

Example:

```
SecRule &REQUEST_HEADERS:Host "@eq 0" \
    "log,id:60008,severity:2,msg:'Request Missing a Host Header'"
```

Note

These are the reserved ranges:

- 1-99,999; reserved for local (internal) use. Use as you see fit but do not use this range for rules that are distributed to others.
- 100,000-199,999; reserved for internal use of the engine, to assign to rules that do not have explicit IDs.
- 200,000-299,999; reserved for rules published at modsecurity.org.
- 300,000-399,999; reserved for rules published at gotroot.com.
- 400,000-419,999; unused (available for reservation).
- 420,000-429,999; reserved for ScallyWhack [http://projects.otaku42.de/wiki/ScallyWhack].
- 430,000-899,999; unused (available for reservation).
- 900,000-999,999; reserved for the Core Rules [http://www.modsecurity.org/projects/rules/] project.
- 1,000,000 and above; unused (available for reservation).

initcol

Description: Initialises a named persistent collection, either by loading data from storage or by creating a new collection in memory.

Action Group: Non-Disruptive

Example: The following example initiates IP address tracking.

```
SecAction initcol:ip=%{REMOTE_ADDR},nolog
```

Note

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

1. `CREATE_TIME` - date/time of the creation of the collection.
2. `KEY` - the value of the initcol variable (the client's IP address in the example).
3. `LAST_UPDATE_TIME` - date/time of the last update to the collection.
4. `TIMEOUT` - date/time in seconds when the collection will be updated on disk from memory (if no other updates occur).
5. `UPDATE_COUNTER` - how many times the collection has been updated since creation.
6. `UPDATE_RATE` - is the average rate updates per minute since creation.

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

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

Action Group: Non-Disruptive

Example:

```
SecAction initcol:ip=%{REMOTE_ADDR},log
```

Note

This action will log matches to the Apache error log file and the ModSecurity audit log.

msg

Description: Assigns a custom message to the rule or chain.

Action Group: Metadata

Example:

```
SecRule &REQUEST_HEADERS:Host "@eq 0" \
    "log,id:60008,severity:2,msg:'Request Missing a Host Header'"
```

Note

The msg information appears in the error and/or audit log files and is not sent back to the client in re-

sponse headers.

multiMatch

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

Action Group: Non-Disruptive

Example:

```
SecDefaultAction log,deny,phase:1,t:removeNulls,t:lowercase
SecRule ARGS "attack" multiMatch
```

Note

Normally, variables are evaluated once, only after all transformation functions have completed. With multiMatch, variables are checked against the operator before and after every transformation function that changes the input.

noauditlog

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

Action Group: Non-Disruptive

Example:

```
SecRule REQUEST_HEADERS:User-Agent "Test" allow,noauditlog
```

Note

If the SecAuditEngine is set to On, all of the transactions will be logged. If it is set to RelevantOnly, then you can control it with the noauditlog action. Even if the noauditlog action is applied to a specific rule and a rule either before or after triggered an audit event, then the tranaction will be logged to the audit log. The correct way to disable audit logging for the entire transaction is to use "`ctl:auditEngine=Off`"

nolog

Description: Prevents rule matches from appearing in both the error and audit logs.

Action Group: Non-Disruptive

Example:

```
SecRule REQUEST_HEADERS:User-Agent "Test" allow,nolog
```

Note

The nolog action also implies noauditlog.

pass

Description: Continues processing with the next rule in spite of a successful match.

Action Group: Disruptive

Example:

```
SecRule REQUEST_HEADERS:User-Agent "Test" log,pass
```

Note

Transaction will not be interrupted but it will be logged (unless logging has been suppressed).

pause

Description: Pauses transaction processing for the specified number of milliseconds.

Action Group: Disruptive

Example:

```
SecRule REQUEST_HEADERS:User-Agent "Test" log,deny,status:403,pause:5000
```

Note

This feature can be of limited benefit for slowing down Brute Force Scanners, however use with care. If you are under a Denial of Service type of attack, the pause feature may make matters worse as this feature will cause child processes to sit idle until the pause is completed.

phase

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

Action Group: Disruptive

Example:

```
SecDefaultAction log,deny,phase:1,t:removeNulls,t:lowercase
SecRule REQUEST_HEADERS:User-Agent "Test" log,deny,status:403
```

Note

Keep in mind that is you specify the incorrect phase, the target variable that you specify may be empty. This could lead to a false negative situation where your variable and operator (RegEx) may be correct, but it misses malicious data because you specified the wrong phase.

proxy

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

Action Group: Disruptive

Example:

```
SecRule REQUEST_HEADERS:User-Agent "Test" log,proxy:http://www.honeypothost.com/
```

Note

For this action to work, mod_proxy must also be installed. This action is useful if you would like to proxy matching requests onto a honeypot webserver.

redirect

Description: Intercepts transaction by issuing a redirect to the given location.

Action Group: Disruptive

Example:

```
SecRule REQUEST_HEADERS:User-Agent "Test" \
    log,redirect:http://www.hostname.com/failed.html
```

Note

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

Description: Specifies rule revision.

Action Group: Metadata

Example:

```
SecRule REQUEST_METHOD "^PUT$" "id:340002,rev:1,severity:2,msg:'Restricted HTTP function'"
```

Note

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

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

Action Group: Non-Disruptive

Example:

```
SecAction nolog,phase:2,sanitiseArg:password
```

Note

The sanitize actions do not sanitize any data within the actual raw requests but only on the copy of data within memory that is set to log to the audit log. It will not sanitize the data in the modsec_debug.log file

(if the log level is set high enough to capture this data).

sanitiseMatched

Description: Sanitises the variable (request argument, request header, or response header) that caused a rule match.

Action Group: Non-Disruptive

Example: 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
```

Note

Same note as sanitiseArg.

sanitiseRequestHeader

Description: Sanitises a named request header.

Action Group: Non-Disruptive

Example: This will sanitise the data in the Authorization header.

```
SecAction log,phase:1,sanitiseRequestHeader:Authorization
```

Note

Same note as sanitiseArg.

sanitiseResponseHeader

Description: Sanitises a named response header.

Action Group: Non-Disruptive

Example: This will sanitise the Set-Cookie data sent to the client.

```
SecAction log,phase:3,sanitiseResponseHeader:Set-Cookie
```

Note

Same note as sanitiseArg.

severity

Description: Assigns severity to the rule it is placed with.

Action Group: Metadata

Example:

```
SecRule REQUEST_METHOD "^PUT$" "id:340002,rev:1,severity:2,msg:'Restricted HTTP function'"
```

Note

The severity numbers follow the Syslog convention:

- 0 = EMERGENCY
- 1 = ALERT
- 2 = CRITICAL
- 3 = ERROR
- 4 = WARNING
- 5 = NOTICE
- 6 = INFO
- 7 = DEBUG

setuid

Description: Special-purpose action that initialises the `USER` collection.

Action Group: Non-Disruptive

Example:

```
SecAction setuid:%{REMOTE_USER},nolog
```

Note

After initialisation takes place the variable `USERID` will be available for use in the subsequent rules.

setsid

Description: Special-purposeaction that initialises the `SESSION` collection.

Action Group: Non-Disruptive

Example:

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

Note

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.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

Description: Creates, removes, or updates an environment variable.

Action Group: Non-Disruptive

Examples:

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

```
setenv:name=value
```

To remove a variable:

```
setenv:!name
```

Note

This action can be used to establish communication with other Apache modules.

setvar

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

Action Group: Non-Disruptive

Examples:

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

Description: Skips one or more rules (or chains) on successful match.

Action Group: Non-Disruptive

Example:

```
SecRule REQUEST_URI "^/$" "chain,skip:2"
SecRule REMOTE_ADDR "^127\.0\.0\.1$" "chain"
SecRule REQUEST_HEADERS:User-Agent "^Apache \(internal dummy connection\)$" "t:none"
SecRule &REQUEST_HEADERS:Host "@eq 0" \
    "deny,log,status:400,id:960008,severity:4,msg:'Request Missing a Host Header'"
SecRule &REQUEST_HEADERS:Accept "@eq 0" \
    "log,deny,log,status:400,id:960015,msg:'Request Missing an Accept Header'"
```

Note

Skip only applies to the current processing phase and not necessarily the order in which the rules appear in the configuration file. If you group rules by processing phases, then skip should work as expected. 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.

status

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

Action Group: Disruptive

Example:

```
SecDefaultAction log,deny,status:403,phase:1
```

Note

Staus actions defined in Apache scope locations (such as Directory, Location, etc...) may be superceded by phase:1 action settings. 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.

t

Description: 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.

Action Group: Non-Disruptive

Example:

```
SecDefaultAction log,deny,phase:1,t:removeNulls,t:lowercase
SecRule REQUEST_COOKIES:SESSIONID "47414e81cbbef3cf8366e84eeacba091" \
    log,deny,status:403,t:md5
```

Note

Any transformation functions that you specify in a SecRule will be in addtion to previous ones specified in SecDefaultAction. Use of "t:none" will remove all transformation functions for the specified rule.

xmlns

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

Action Group: Non-Disruptive

Example:

```
SecRule REQUEST_HEADERS:Content-Type "text/xml" \
    phase:1,pass,ctl:requestBodyProcessor=XML,ctl:requestBodyAccess=On,xmlns:xsd="http://ww
SecRule XML:/soap:Envelope/soap:Body/q1:getInput/id() "123" phase:2,deny
```

Operators

A number of operators can be used in rules, as documented below. The operator syntax used the "@" symbol followed by the specific operator name.

eq

Description: This operator is a numerical comparison and stands for "equal to."

Example:

```
SecRule &REQUEST_HEADERS_NAMES "@eq 15"
```

ge

Description: This operator is a numerical comparison and stands for "greater than or equal to."

Example:

```
SecRule &REQUEST_HEADERS_NAMES "@ge 15"
```

gt

Description: This operator is a numerical comparison and stands for "greater than."

Example:

```
SecRule &REQUEST_HEADERS_NAMES "@gt 15"
```

inspectFile

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

Example:

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

le

Description: This operator is a numerical comparison and stands for "less than or equal to."

Example:

```
SecRule &REQUEST_HEADERS_NAMES "@le 15"
```

lt

Description: This operator is a numerical comparison and stands for "less than."

Example:

```
SecRule &REQUEST_HEADERS_NAMES "@lt 15"
```

rbl

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

Example:

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

rx

Description: Regular expression operator. This is the default operator, so if the "@" operator is not defined, it is assumed to be rx.

Example:

```
SecRule REQUEST_HEADERS:User-Agent "@rx nikto"
```

Note

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 `(?i)` modificator, as allowed by PCRE.
3. The `PCRE_DOTALL` and `PCRE_DOLLAR_ENDONLY` flags are set during compilation, meaning a single dot will match any character, including the newlines and a `$` end anchor will not match a trailing newline charater.

validateByteRange

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

Example:

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

Note

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). 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.

validateByteRange is similar to the ModSecurity 1.X SecFilterForceByteRange Directive however since it works in a rule context, it has the following differences:

- You can specify a different range for different variables.
- It has an "event" context (id, msg....)
- It is executed in the flow of rules rather than being a built in pre-check.

validateDTD

Description: This operator requires the request body to be processed as XML.

Example:

```
SecDefaultAction log,deny,status:403,phase:2
SecRule REQUEST_HEADERS:Content-Type ^text/xml$ \
    phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML
SecRule REQBODY_PROCESSOR "!^XML$" nolog,pass,skip:1
SecRule XML "@validateDTD /path/to/apache2/conf/xml.dtd"
```

validateSchema

Description: This operator requires the request body to be processed as XML.

Example:

```
SecDefaultAction log,deny,status:403,phase:2
SecRule REQUEST_HEADERS:Content-Type ^text/xml$ \
    phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML
SecRule REQBODY_PROCESSOR "!^XML$" nolog,pass,skip:1
SecRule XML "@validateSchema /path/to/apache2/conf/xml.xsd"
```

This operator requires request body to be processed as XML.

validateUrlEncoding

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

Example:

```
SecRule ARGS "@validateUrlEncoding"
```

Note

URL encoding is an HTTP standard for encoding byte values within a URL. The byte is escaped with a % followed by two hexadecimal values (0-F). 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.

validateUtf8Encoding

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

Example:

```
SecRule ARGS "@validateUtf8Encoding"
```

Note

UTF-8 encoding is valid on most web servers. Integer values between 0-65535 are encoded in a UTF-8 byte sequence that is escaped by percents. The short form is two bytes in length.

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).

Miscellaneous Topics

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. In such cases it may be possible to design payload that will be interpreted in one way by one device and in another by the other device. This problem is better known as Impedance Mismatch. It can be exploited to evade the security devices.

While we will continue to enhance ModSecurity to deal with various evasion techniques the problem can only be minimized, but never solved. With so many different application backends chances are some will always do something completely unexpected. The only solution is to be aware of the technologies in the backend when writing rules, adapting the rules to remove the mismatch. See the next section for some examples.

PHP Peculiarities for ModSecurity Users

When writing rules to protect PHP applications you need to pay attention to the following facts:

1. When "register_globals" is set to "On" request parameters are automatically converted to script variables. In some PHP versions it is even possible to override the $GLOBALS array.
2. Whitespace at the beginning of parameter names is ignored. (This is very dangerous if you are writing rules to target specific named variables.)
3. The remaining whitespace (in parameter names) is converted to underscores. The same applies to dots and to a "[" if the variable name does not contain a matching closing bracket. (Meaning that if you want to exploit a script through a variable that contains an underscore in the name you can send a parameter with a whitespace or a dot instead.)
4. Cookies can be treated as request parameters.
5. The discussion about variable names applies equally to the cookie names.
6. 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).
7. When "magic_quotes_gpc" is set to "On" PHP will use backslash to escape the following characters: single quote, double quote, backslash, and the nul byte.
8. 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. The "magic_quotes_sybase" setting completely overrides the "magic_quotes_gpc" behaviour but "ma-

gic_quotes_gpc" still must be set to "On" for the Sybase-specific quoting to be work.

9. PHP will also automatically create nested arrays for you. For example "p[x][y]=1" results in a total of three variables.