MODSECURITY HANDBOOK

The Complete Guide to Securing Your Web Applications

Preview Release

Ivan Ristic

Feisty Duck

ModSecurity Handbook

Ivan Ristić

ModSecurity Handbook

by Ivan Ristić

Table of Contents

Preface

[...]

Audience

[...]

Contents of This Book

[...]

Updates

[...]

Conventions

[...]

Acknowledgements

[...]

Part I: User Guide

1 Introduction

ModSecurity is a tool that will help you secure your web applications. No, scratch that. Actually, ModSecurity is a tool that will help you sleep better at night, and I will explain how. I usually call ModSecurity a *web application firewall (WAF)*, because that's the generally accepted term to refer to the class of products that are specifically designed to secure web applications. Other times I will call it an *HTTP intrusion detection tool*, because I think that name better describes what ModSecurity does. Neither name is entirely adequate, yet we don't have a better one. Besides, it doesn't really matter what we call it. The point is that web applications—yours, mine, everyone's—are terribly insecure on average. We struggle to keep up with the security issues and need any help we can get to secure them.

The idea to write ModSecurity came to me during one of my sleepless nights—I couldn't sleep because I was responsible for the security of several web-based products. I could see the web application security storm on the horizon. (We were then largely in the age of innocence when it came to web application security.) I could see how most web applications were just slapped together with little time spent on design and little time spent on understanding the security issues. Furthermore, not only were web applications insecure, but we had no idea how insecure they were or if they were being attacked. Our only eyes were the web server access and error logs, and they didn't say much.

Which brings me to my point. ModSecurity will help you sleep better at night because, above all, it solves the visibility problem: it lets you see your web traffic. That visibility is key to security: once you are able to see HTTP traffic, you are able to analyze it in real time, record it as necessary, and react to the events. The best part of this concept is that you get to do all of that without actually touching web applications. Even better, the concept can be applied to any application—even the one to which you don't have access to the source code.

There are four guiding principles on which ModSecurity is based, as follows:

Flexibility
: I think that it's fair to say that I built ModSecurity for myself: a security expert who needs to intercept, analyze, and store HTTP traffic. I didn't see much value in hard-

coded functionality, because real life is so complex that everyone needs to do things just slightly differently. ModSecurity achieves flexibility by giving you a powerful rule language, which allows you to do exactly what you need to, in combination with the ability to apply rules only where you need to.

Passiveness
: ModSecurity will take great care to never interact with a transaction unless you tell it to. That is simply because I didn't trust a tool, even the one I built, to make decisions for me. That's why ModSecurity will give you plenty of information, but ultimately leave the decisions to you.

Predictability
: There's no such thing as a perfect tool, but a predictable one is the next best thing. Armed with all the facts, which you will find in the reference manual, the posts on the ModSecurity Blog, or here, you can understand ModSecurity's weak points and work around them.

Feature quality over quantity
: Over the course of six years spent working on ModSecurity, we came up with many ideas for what ModSecurity could do. We didn't act on most of them. We kept them for later. Why? Because we understood that we have limited resources available at our disposal and that our minds (ideas) are far faster than our implementation abilities. We chose to limit the available functionality, but do really well at what we decided to keep in.

There are bits in ModSecurity that fall outside the scope of these four principles. For example, ModSecurity can change the way Apache identifies itself to the outside world, confine the Apache process within a jail, and even implement an elaborate scheme to deal with an once-infamous universal XSS vulnerability in Adobe Reader. Although it was I who added those features, I now think that they detract from the main purpose of ModSecurity, which is a reliable and predictable tool that allows for HTTP traffic inspection.

Brief History

Like many other open source projects, ModSecurity started out as a hobby. Software development had been my primary concern back in 2002, when I realized that producing secure web applications is virtually impossible. As a result, I started to fantasize about a tool that would sit in front of web applications and control the flow of data in and out. The first version was released in November 2002, but a few more months were needed before the tool became useful. Other people started to learn about it, and the popularity of ModSecurity started to rise.

Initially, most of my work was spent wrestling with Apache to gain access to request bodies, which was a crucial ability. Apache 1.3.x did not have any interception or filtering APIs, but I was able to trick it into submission. Apache 2.x improved things by providing APIs that do allow content interception, but there was no documentation to speak of. Nick Kew's excellent *The Apache Modules Book* (Prentice Hall) came out in 2007, by which time it was too late.

By 2004, I was a changed man. Once primarily a software developer, I became obsessed with web application security and wanted to spend more time working on it. I quit my job and started treating ModSecurity as a business. My big reward came in the summer of 2006, when ModSecurity went head to head with other web application firewalls, in an evaluation conducted byForrester Research, and came out very favorably. Later that year, my company was acquired by Breach Security. A team of one eventually become a team of many: Brian Rectanus came to work on ModSecurity, Ofer Shezaf took on the rules, and Ryan C. Barnett the community management and education. ModSecurity 2.0, a complete rewrite, was released in late 2006. At the same time we released ModSecurity Community Console, which combined the functionality of a remote logging sensor and a monitoring and reporting GUI.

The last major update of ModSecurity was 2.5, released in February 2008. There have been 11 maintenance releases since. ModSecurity 2.5.11 (released in November 2009) is the most recent version at the time of writing.

Since leaving Breach Security in January 2009, I am no longer in charge of ModSecurity, but it remains in the capable hands of Brian Rectanus and Ryan C. Barnett. I remain involved and contribute from time to time.

Understanding ModSecurity

ModSecurity is a hybrid web application firewall that relies on the host web server for some of the work. The only supported web server at the moment is Apache 2.x, but it is possible, in principle, to integrate ModSecurity with any other web server that provides sufficient integration APIs.

Apache does for ModSecurity what it does for all other modules—it handles the infrastructure tasks:

1. Decrypts SSL
2. Breaks up the inbound connection stream into HTTP requests
3. Partially parses HTTP requests

4. Invokes ModSecurity, choosing the correct configuration context (`<VirtualHost>`, `<Location>`, etc.)
5. De-chunks request bodies as necessary

There a few additional tasks Apache performs in a reverse proxy scenario:

1. Forwards requests to backend servers (with or without SSL)
2. Partially parses HTTP responses
3. De-chunks response bodies as necessary

The advantage of a hybrid implementation is that it is very efficient—the duplication of work is minimal when it comes to HTTP parsing. A couple of disadvantages of this approach are that you don't always get access to the raw data stream and that web servers sometimes don't process data in a way a security-conscious tool would. In the case of Apache, the hybrid approach works reasonably well, with a few minor issues:

Request line and headers are NUL-terminated
: This is normally not a problem, because what Apache doesn't see cannot harm any module or application. In some very rare cases, however, the purpose of the NUL-byte evasion is to hide things and this Apache behavior only helps with the hiding.

Request header transformation
: Apache will canonicalize request headers, combining multiple headers that use the same name and collapsing those that span two or more lines. The transformation may make it difficult to detect subtle signs of evasion, but in practice this hasn't been a problem yet.

Quick request handling
: Apache will handle some requests quickly, leaving ModSecurity unable to do anything but notice them in the logging phase. Invalid HTTP requests, in particular, will be rejected by Apache without ModSecurity having a say.

No access to some response headers
: Because of the way Apache works, the `Server` and `Date` response headers are invisible to ModSecurity; they cannot be inspected or logged.

What ModSecurity Does

The functionality offered by ModSecurity falls roughly into four areas:

Parsing
: ModSecurity tries to make sense of as much data as available. The supported data formats are backed by security-conscientious parsers that extract bits of data and store them for the use in the rules.

Buffering
: In a typical installation, both request and response bodies will be buffered. This means that ModSecurity usually sees complete requests before they are passed to application for processing, and complete responses before they are sent to clients. Buffering is an important feature, because it is the only way to provide reliable blocking. The downside of buffering is that it requires additional RAM to store the request and response body data.

Logging
: Full transaction logging (also referred to as *audit logging*) is a big part of what ModSecurity does. This feature allows you to record complete HTTP traffic, instead of just rudimentary access log information. Request headers, request body, response header, response body—all those bits will be available to you. It is only with the ability to see what is happening that you will be able to stay in control.

Rule Engine
: The rule engine builds on the work performed by all other components. By the time the rule engine starts operating, the various bits and pieces of data it requires will all be prepared and ready for inspection. At that point, the rules will take over to assess the transaction and take actions as necessary.

> **Note**
>
> There's one thing ModSecurity purposefully avoids to do: as a matter of design, ModSecurity does not support data sanitization. I don't believe in sanitization, purely because I believe that it is too difficult to get right. If you know for sure that you are being attacked (as you have to before you can decide to sanitize), then you should refuse to process the offending requests altogether. Attempting to sanitize merely opens a new battlefield where your attackers don't have anything to lose, but everything to win. You, on the other hand, don't have anything to win, but everything to lose.

What Rules Look Like

Everything in ModSecurity revolves around two things: configuration and rules. The configuration tells ModSecurity how to process the data it sees; the rules decide what to do with the processed data. Although it is too early to go into how the rules work, I will show you a quick example here just to give you an idea what they look like.

For example:

```
SecRule ARGS "<script>" log,deny,status:404
```

Even without further assistance, you can probably recognize the part in the rule that specifies what we wish to look for in input data (`<script>`). Similarly, you will easily figure out what will happen if we do find the desired pattern (`log,deny,status:404`). Things will become more clear if I tell you about the general rule syntax, which is the following:

```
SecRule VARIABLES OPERATOR ACTIONS
```

The three parts have the following meanings:

1. The `VARIABLES` part tells ModSecurity where to look. The `ARGS` variable, used in the example, means all request parameters.
2. The `OPERATOR` part tells ModSecurity how to look. In the example, we have a regular expression pattern, which will be matched against `ARGS`.
3. The `ACTIONS` part tells ModSecurity what to do on a match. The rule in the example gives three instructions: `log` problem, `deny` transaction and use the status 404 for the denial (`status:404`).

I hope you are not dissapointed with the simplicity of this first rule. I promise you that by combining the various facilities offered by ModSecurity, you will be able to write very useful rules that implement complex logic where necessary.

Transaction Lifecycle

In ModSecurity, every transaction goes through 5 steps, or phases. In each of the phases, ModSecurity will do some work at the beginning (e.g., parse data that has become avilable), invoke the rules specified to work in that phase, and perhaps do a thing or two after the phase rules have finished. At first glance, it may seem that 5 phases are too many, but there's a reason why each of the phases exist. There is always one thing, sometimes several, that can only be done at a particular moment in the transaction lifecycle.

Request Headers (1)

The request headers phase is the first entry point for ModSecurity. The principal purpose of this phase is to allow rule writers to assess a request before the costly request body processing is undertaken. Similarly, there is often a need to influence how ModSecurity will process a request body, and this phase is the place to do it. For example, ModSecurity will not parse an XML request body by default, but you can instruct it do so by placing the appropriate rules into phase 1. (If you care about XML processing, it is described in detail in Chapter 12, *Handling XML*).

Request Body (2)
: The request body phase is the main request analysis phase and takes place immediatelly after a complete request body has been received and processed. The rules in this phase have all the available request data at their disposal.

Response Headers (3)
: The response headers phase takes place after response headers become available, but before a response body is read. The rules that need to decide whether to inspect a response body should run in this phase.

Response Body (4)
: The response body phase is the main response analysis phase. By the time this phase begins, the response body will have been read, with all its data available for the rules to make their decisions.

Logging (5)
: The logging phase is special, in more ways than one. First, it's the only phase from which you cannot block. By the time this phase runs, the transaction will have finished, so there's little you can do but record the fact that it happened. Rules in this phase are run to control how logging is done.

Lifecycle Example

To give you a better idea how what happens on every transaction, we'll examine a detailed debug log of one POST transaction. I've deliberately chosen a transaction type that uses the request body as a principal method to transmit data, because following such a transaction will exercise most parts of ModSecurity. To keep things relatively simple, I used a configuration without any rules, removed some of the debug log lines for clarity, and removed the timestamps and some additional metadata from each line.

> **Note**
>
> Please do not try to understand everything about the logs at this point. The idea is just to get a general feel about how ModSecurity works, and to introduce you to debug logs. Very quickly after starting to use ModSecurity, you will discover that the debug logs will be an indespensable rule writing and troubleshooting tool.

The transaction I am using as an example in this section is very straightforward. I made a point of placing request data in two different places, parameter *a* in the query string and parameter *b* in the request body, but there is little else of interest in the request:

```
POST /?a=test HTTP/1.0
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 6

b=test
```

The response is entirely unremarkable:

```
HTTP/1.1 200 OK
Date: Sun, 17 Jan 2010 00:13:44 GMT
Server: Apache
Content-Length: 12
Connection: close
Content-Type: text/html

Hello World!
```

ModSecurity is first invoked by Apache after request headers become available, but before a request body (if any) is read. First comes the initialization message, which contains the unique transaction ID generated by `mod_unique_id`. Using this information, you should be able to pair the information in the debug log with the information in your access and audit logs. At this point, ModSecurity will parse the information on the request line and in the request headers. In this example, the query string part contains a single parameter (`a`), so you will see a message documenting its discovery. ModSecurity will then create a transaction context and invoke the `REQUEST_HEADERS` phase:

```
[4] Initialising transaction (txid SopXW38EAAE9YbLQ).
[5] Adding request argument (QUERY_STRING): name "a", value "test"
[4] Transaction context created (dcfg 8121800).
[4] Starting phase REQUEST_HEADERS.
```

Assuming that a rule didn't block the transaction, ModSecurity will now return control to Apache, allowing other modules to process the request before control is given back to it.

In the second phase, ModSecurity will first read and process the request body, if it is present. In the following example, you can see three messages from the input filter, which tell you what was read. The fourth message tells you that one parameter was extracted from the request body. The content type used in this request (`application/x-www-form-urlencoded`) is one of the types ModSecurity recognizes and parses automatically. Once the request body is processed, the `REQUEST_BODY` rules are processed.

```
[4] Second phase starting (dcfg 8121800).
[4] Input filter: Reading request body.
[9] Input filter: Bucket type HEAP contains 6 bytes.
[9] Input filter: Bucket type EOS contains 0 bytes.
[5] Adding request argument (BODY): name "b", value "test"
[4] Input filter: Completed receiving request body (length 6).
[4] Starting phase REQUEST_BODY.
```

The filters that keep being mentioned in the logs are parts of ModSecurity that handle request and response bodies:

```
[4] Hook insert_filter: Adding input forwarding filter (r 81d0588).
[4] Hook insert_filter: Adding output filter (r 81d0588).
```

There will be a message in the debug log every time ModSecurity sends a chunk of data to the request handler, and one final message to say that there isn't any more data in the buffers.

```
[4] Input filter: Forwarding input: mode=0, block=0, nbytes=8192…
(f 81d2228, r 81d0588).
[4] Input filter: Forwarded 6 bytes.
[4] Input filter: Sent EOS.
[4] Input filter: Input forwarding complete.
```

Shortly thereafter, the output filter will start receiving data, at which point the `RESPONSE_HEADERS` rules will be invoked:

```
[9] Output filter: Receiving output (f 81d2258, r 81d0588).
[4] Starting phase RESPONSE_HEADERS.
```

Once all the rules have run, ModSecurity will continue to store the response body in its buffers, after which it will run the `RESPONSE_BODY` rules:

```
[9] Output filter: Bucket type MMAP contains 12 bytes.
[9] Output filter: Bucket type EOS contains 0 bytes.
[4] Output filter: Completed receiving response body (buffered full - 12 bytes).
[4] Starting phase RESPONSE_BODY.
```

Again, assuming that none of the rules blocked, the accumulated response body will be forwarded to the client:

```
[4] Output filter: Output forwarding complete.
```

Finally, the logging phase will commence. The `LOGGING` rules will be run first to allow them to influence logging, after which the audit logging subsystem will be invoked to log the transaction if necessary. A message from the audit logging subsystem will be the last transaction message in the logs. In this example, ModSecurity tells us that it didn't find anything of interest in the transaction and that it sees no reason to log it:

```
[4] Initialising logging.
[4] Starting phase LOGGING.
[4] Audit log: Ignoring a non-relevant request.
```

File Upload Example

Handling of file uploads changes how requests are processed. The changes can be best understood by again following the activity in the debug log:

```
[4] Input filter: Reading request body.
[9] Multipart: Boundary: --------------------------2411583925858
[9] Input filter: Bucket type HEAP contains 256 bytes.
[9] Multipart: Added part header "Content-Disposition" "form-data; name=\"f\";…
filename=\"eicar.com.txt\""
[9] Multipart: Added part header "Content-Type" "text/plain"
[9] Multipart: Content-Disposition name: f
[9] Multipart: Content-Disposition filename: eicar.com.txt
[4] Multipart: Created temporary file:…
/opt/modsecurity/var/tmp/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF
[9] Multipart: Changing file mode to 0600:…
/opt/modsecurity/var/tmp/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF
[9] Multipart: Added file part 9c870b8 to the list: name "f" file name…
"eicar.com.txt" (offset 140, length 68)
[9] Input filter: Bucket type EOS contains 0 bytes.
[4] Reqest body no files length: 96
[4] Input filter: Completed receiving request body (length 256).
```

In addition to seeing the multipart parser in action, you see ModSecurity creating a temporary file (into which it will extract the upload) and adjusting its privileges to match the desired configuration.

Then, at the end of the transaction, you will see the cleanup and the temporary file deleted:

```
[4] Multipart: Cleanup started (remove files 1).
[4] Multipart: Deleted file (part)…
"/opt/modsecurity/var/tmp/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF"
```

The temporary file will not be deleted if ModSecurity decides to keep an uploaded file. Instead, it will be moved to the storage area:

```
[4] Multipart: Cleanup started (remove files 0).
[4] Input filter: Moved file from…
"/opt/modsecurity/var/tmp/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF" to…
"/opt/modsecurity/var/upload/20090819-175503-SowuZ38AAQEAACV-Agk-file-gmWmrF".
```

In the example traces, you've observed an upload of a small file that was stored in RAM. When large uploads take place, ModSecurity will attempt to use RAM at first, switching to on-disk storage once it becomes obvious that the file is larger:

```
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
```

```
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 1536 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 576 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[9] Input filter: Bucket type HEAP contains 8000 bytes.
[4] Input filter: Request too large to store in memory, switching to disk.
```

A new file will be created to store the entire raw request body:

```
[4] Input filter: Created temporary file to store request body:…
/opt/modsecurity/var/tmp//20090819-180105-SowvOX8AAQEAACWAArs-request_body-4nZjqf
[4] Input filter: Wrote 129559 bytes from memory to disk.
```

This file is always deleted in the cleanup phase:

```
[4] Input filter: Removed temporary file:…
/opt/modsecurity/var/tmp//20090819-180105-SowvOX8AAQEAACWAArs-request_body-4nZjqf
```

Impact of ModSecurity on Web Server

The addition of ModSecurity will change how your web server operates. You pay for the additional flexibility and security with the increased CPU and RAM consumption on the server where ModSecurity runs. Following is a detailed list of the various activities that increase resource consumption:

- ModSecurity will add to the parsing already done by Apache, and that results in a slight increase of CPU consumption.
- Complex parsers (e.g., XML) are more expensive.
- The handling of file uploads may require I/O operations. In some cases, inbound data will be duplicated on disk.
- The parsing will add to the RAM consumption, because every extracted element (e.g., a request parameter) will need to be copied into its own space.
- Request bodies and response bodies are usually buffered in order to support reliable blocking.

- Every rule in your configuration will use some of the CPU time (for the operator) and RAM (to transform input data before it can be analyzed).
- Some of the operators used in the rules (e.g., the regular expression operator) are CPU-intensive.
- Full transaction logging is an expensive I/O operation.

In practice, this list is mostly important to keep you informed; what matters is that you have enough resources to support your ModSecurity needs. If you do, then it doesn't matter how expensive ModSecurity is. Also, what's expensive to someone may not be to someone else. If you don't have enough resources to do everything you want with ModSecurity, you will need to monitor the operation of your system and remove some of the functionality to reduce the resource consumption. Virtually everything that ModSecurity does is configurable, so you should have no problems doing that.

It is generally easier to run ModSecurity in reverse proxy mode, because then you usually have an entire server (with its own CPU and RAM) to play with. In embedded mode, ModSecurity will add to the processing already done by the web server, so this method is more challenging on a busy server.

For what it's worth, ModSecurity generally uses the minimal necessary resources to perform the desired functions, so this is really a case of exchanging functionality for speed: if you want to do more, you have to pay more.

Embedded vs. Reverse Proxy Mode

ModSecurity generally cares doesn't care whether it is deployed in embedded or reverse proxy mode. In the reverse proxy mode, Apache takes care of the transfer of data to the backend server and back, so there is very little for ModSecurity to worry about. There are only few small differences, which I am listing here for reference:

1. In an embedded scenario, there will typically be a resource (a script or a file) that is used to fulfill each request. ModSecurity rules can inspect the properties of such files (the `SCRIPT_*` family of variables allows access). In the reverse proxy mode, virtually all requests will be fullfilled by backend servers, which means that local resources won't be used and that the use of the variables that reference them makes little sense.
2. When embedded, ModSecurity gives access to the web server envinronment and error log. When used in a reverse proxy, you still get access to both the environment and the error log, but to those of the reverse proxy. The backend servers will have their own environments and error logs, which ModSecurity can't access.

3. Apache's <Directory>, <DirectoryMatch>, <Files> and <FileMatch> configuration contexts never match when used in a reverse proxy.
4. There are potential evasion issues when a reverse proxy is used in front of a back-end system that interprets URIs differently (e.g., if you have a Unix box in front of a Windows box). In such cases, you have to be very careful if you're using the <Location> configuration context. The <Location> configuration context is case-sensitive and recognizes only forward slashes, whereas other platforms may have filesystems that are case-insensitive, or web servers that support the backslash as the URI path separator.

Table 1.1. Variables sensitive to operating mode

Variable	Availability in reverse proxy mode
AUTH_TYPE	Reverse proxy authentication
PATH_INFO	Not available
ENV	Reverse proxy environment
SCRIPT_BASENAME	Not available
SCRIPT_FILENAME	Not available
SCRIPT_GID	Not available
SCRIPT_GROUPNAME	Not available
SCRIPT_MODE	Not available
SCRIPT_UID	Not available
SCRIPT_USERNAME	Not available
SERVER_ADDR	Reverse proxy address
SERVER_NAME	Reverse proxy name
SERVER_PORT	Reverse proxy port
WEBSERVER_ERROR_LOG	Reverse proxy error log

Missing from ModSecurity

ModSecurity is a very good tool, but there are a number of features, big and small, that could be added. The small features are those that would make your life with ModSecurity easier, perhaps automating some of the boring work (e.g., persistent blocking, which you now have to do manually). But there are really only two features that I would call "missing" (assuming you accept that a GUI is not within the scope of the project):

Learning

Defending web applications is difficult, because there are so many of them and they are all different. (I often say that every web application effectively creates its own communication protocol.) It would be very handy to have ModSecurity observe application traffic and create a model that could later be used to generate policy or assist with false positives. While I was at Breach Security, I started a project call ModProfiler [http://www.modsecurity.org/projects/modprofiler/] as a step toward learning, but that project is still as I left it, as version 0.2.

Passive mode of deployment

ModSecurity can be embedded only in Apache 2.x, but when you deploy it as a reverse proxy, it can be used to protect any web server. Reverse proxies are not everyone's cup of tea, however, and sometimes it would be very handy to deploy ModSecurity passively, without having to change anything on the network.

Resources

The most important resource is of course ModSecurity's web site [https://www.modsecurity.org], and you should visit it from time to time, as well as subscribe to receive the updates from the blog. This section contains a list of other useful ModSecurity resources, grouped according to purpose.

Figure 2-1. The homepage of www.modsecurity.org

General Resources

The following resources are the bare essentials:

Official documentation

The official ModSecurity documentation consists of two files: the *Reference Manual*, which covers the Rule Language, and the *Data Formats Guide*, which explains the data formats used for storage and the error messages. To get the most recent version of these two documents, go to https://www.modsecurity.org/documentation/ [http://www.modsecurity.org/documentation/]. Always keep a bookmark to the documentation of the same version of ModSecurity that you are working with. Although the ModSecurity web site keeps the links to the documentation of the most recent release only, the previous versions are all there. To access them you just need to replace the current version number with whatever earlier version number you need.

Issue tracker

The ModSecurity issue tracker [https://www.modsecurity.org/tracker/] is the place you will want visit for one of two reasons: to report a problem with ModSecurity itself (e.g., when you find a bug) or to check out the progress on the next (major or minor) version. Before reporting any problems, go through the Support Checklist [http://www.modsecurity.org/documentation/support-request-checklist.html], which will help you assemble the information required to help resolve your problem. Providing as much information as you can will help the developers understand and replicate the problem, and provide a fix (or a workaround) quickly.

Users' mailing list

The users' mailing list [http://lists.sourceforge.net/lists/listinfo/mod-security-users] (mod-security-users@lists.sourceforge.net) is a general-purpose mailing list where you can discuss ModSecurity. Feel free to ask questions, propose improvements, and discuss ideas. That is the place where you'll hear first about new ModSecurity versions.

ModSecurity@Freshmeat

If you subscribe to the users' mailing list, you will generally find out about new versions of ModSecurity as soon as they are released. If you care only about version releases, however, you may consider subscribing to the new version notifications at the ModSecurity page at Freshmeat [http://freshmeat.net/projects/modsecurity].

Core Rules mailing list

Starting with version 2, the Core Rules [http://www.owasp.org/index.php/Category:OWASP_ModSecurity_Core_Rule_Set_Project] project is part of OWASP [http://www.owasp.org], and has a separate mailing list (owasp-modsecurity-core-rule-set@lists.owasp.org).

Developer Resources

If you are interested in development work, you will need these:

Developers' mailing list

The developers' mailing list [https://lists.sourceforge.net/lists/listinfo/mod-security-developers] is generally a lonely place, but if you do decide to start playing with the ModSecurity source code, this list is the place to go to discuss your work.

Source code access

The source code of ModSecurity is hosted at a Subversion repository at SourceForge [http://sourceforge.net/projects/mod-security/develop], which allows you to access it directly or through a web-based user interface.

FishEye interface

If you are not looking to start developing immediately but still want to have a look at the source code of ModSecurity, I recommend that you use the ModSecurity FishEye interface [https://www.modsecurity.org/fisheye/], which is much better than the stock interface available at SourceForge.

Related Projects

There is a number of projects and sites that are related to ModSecurity in one way or another. The most interesting ones are:

jwall.org

Christian Bockermann runs jwall.org [http://www.jwall.org] and provides several interesting application security projects and ModSecurity tools. The most useful tool is the AuditConsole, a web-based application that consolidates audit logs from multiple ModSecurity sensors.

ModSecurity Community Console

When Breach Security acquired ModSecurity, they decided to give away the then-commercial log consolidation tool for free, calling it ModSecurity Community Console [https://www.modsecurity.org/projects/console/]. The tool has seen little improvement since 2006, but it fulfills the elementary requirements of log aggregation and storage. It is limited to supporting up to three ModSecurity sensors.

REMO

REMO [http://www.netnea.com/cms/?q=remo] (short for Rule Editor for ModSecurity) is a web-based rule editor designed for whitelist creation. It is written by Christian Folini, and uses existing audit logs to assist you in creating a positive-security shield around an application.

WebDefend GEM

WebDefend GEM [http://www.breach.com/products/webdefend-global-event-manager.html] (Global Event Manager) is a commercial web application firewall event management solution from Breach Security that will accept audit logs from ModSecurity sensors, as well as from WebDefend and the Akamai web application firewall.

2 Installation

Before you can install ModSecurity, you need to decide if you want to compile it from source or use a binary version—either one included with your operating system or produced by a third party. Each of the options comes with its advantages and disadvantages, as listed in Table 2.1, "Installation options".

Table 2.1. Installation options

Installation type	Advantages	Disadvantages
Operating system version	• Fully automated installation • Maintenance included	• May not be the latest version
Third-party binary	• Semi-automated installation	• May not be the latest version • Manual download and updates • Do you trust the third party?
Source code	• Can always use the latest version • Can use experimental versions • Can make changes, apply patches, and make emergency security fixes	• Manual installation and maintenance required • A lot of work involved with rolling your own version

In some cases, you won't have a choice. For example, if you've installed Apache from source, you will need to install ModSecurity from source too (you will be able to reuse the system packages, of course). The following questions may help you to make the decision:

- Do you intend to use ModSecurity seriously?
- Are you comfortable compiling programs from source?
- Do you have enough time to spend on the compilation and the successive maintenance of a custom-installed program?
- Will you need to make changes to ModSecurity, or write your own extensions?

I generally try to use binary packages when they are available (and they are available on Debian, which is currently my platform of choice). When I build dedicated reverse proxy

installations, however, I tend to build everything from source, because that allows me access to the latest Apache and ModSecurity versions, and makes it easier to tweak things (by changing the source code of either Apache or ModSecurity) when I want to.

Installation from Source

Installing from source is the preferred approach to installing ModSecurity, mostly because that way you get the latest (and best) version, and because you are able to make any changes you want.

Downloading Releases

To download ModSecurity, go to `http://www.modsecurity.org/download/`. You will need both the main distribution and the cryptographic signature:

```
$ wget http://www.modsecurity.org/download/modsecurity-apache_2.5.10-dev2.tar.gz
$ wget http://www.modsecurity.org/download/modsecurity-apache_2.5.10-dev2.tar.gz.asc
```

Verify the signature before doing anything else. That will ensure that the package you've just downloaded does not contain a trojan horse planted by a third party and that it hasn't been corrupted during transport.

```
$ gpg --verify modsecurity-apache_2.5.10-dev2.tar.gz.asc
gpg: Signature made Wed 12 Aug 2009 23:27:06 BST using DSA key ID E77B534D
gpg: Can't check signature: public key not found
```

Your first attempt may not provide the expected results, but that can be easily solved by importing the referenced key from a key server:

```
$ gpg --recv-keys E77B534D
gpg: requesting key E77B534D from hkp server keys.gnupg.net
gpg: /home/guest/.gnupg/trustdb.gpg: trustdb created
gpg: key E77B534D: public key "Brian Rectanus (work) <brian.rectanus@breach.com>"…
imported
gpg: no ultimately trusted keys found
gpg: Total number processed: 1
gpg:               imported: 1
```

Now you can try again:

```
$ gpg --verify modsecurity-apache_2.5.10-dev2.tar.gz.asc
gpg: Signature made Wed 12 Aug 2009 23:27:06 BST using DSA key ID E77B534D
gpg: Good signature from "Brian Rectanus (work) <brian.rectanus@breach.com>"
gpg:                 aka "Brian Rectanus <brian@rectanus.net>"
```

```
gpg:                 aka "Brian Rectanus (personal) <brectanu@gmail.com>"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the owner.
```

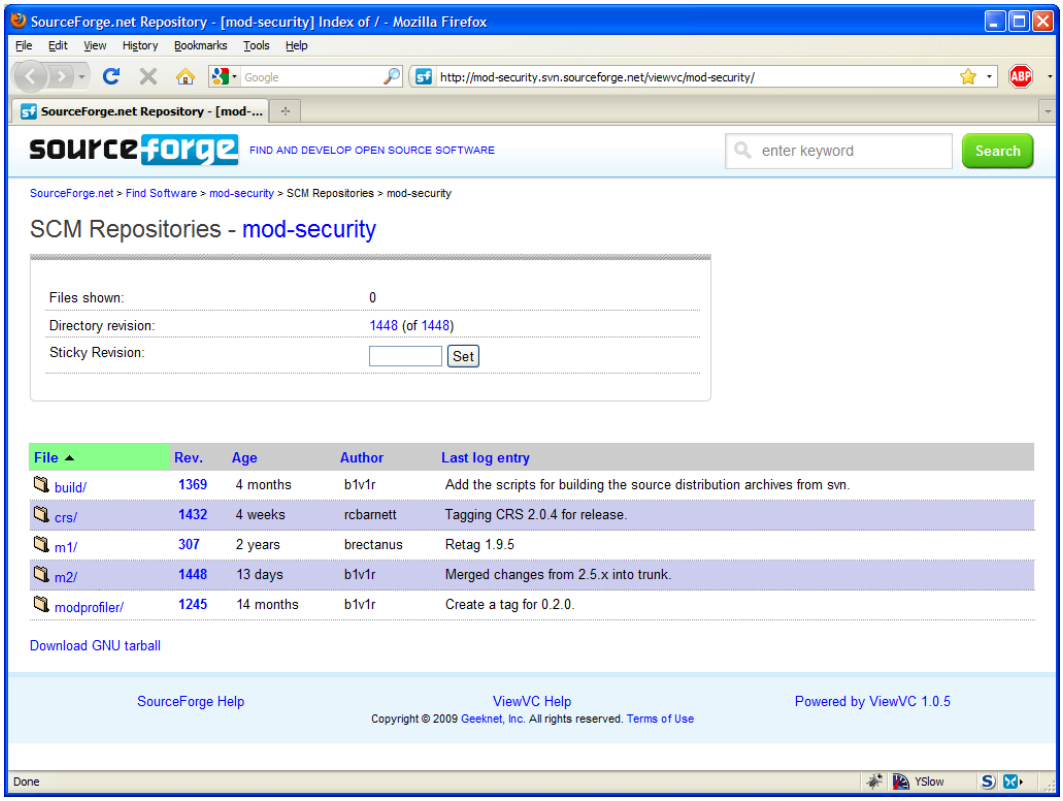
If you don't know what the the previous above warning is about, then I dare say you can safely ignore it. The warning basically tells you that you've downloaded Brian's key from somewhere but that you don't really *know* that it belongs to him. The only way to be sure is to meet Brian in real life, or meet someone else who you can trust who knows him personally. If you want to learn more, look up *web of trust* on Wikipedia.

Downloading from Repository

If you want to be on the cutting edge, downloading the latest development version directly from the Subversion (the source code control system used by the ModSecurity project) repository is the way to go. When you do this, you'll get new features days and even months before they make it into an official stable release. Having said that, however, there is a reason why we call some versions "stable." When you use a repository version of ModSecurity, you need to accept that there is no guarantee whatsoever that it will work correctly. For what it's worth, I am currently running a development version (of a future 2.6.x version) in production, and I am confident that it will not bring my server down.

Before you can install ModSecurity, you need to know exactly where to look. The repository is hosted with SourceForge, and you can browse it by pointing your browser at http://modsecurity.svn.sourceforge.net/viewvc/mod-security/, after which you will see the screen similar to that in Figure 2-1, "ModSecurity repository root".

Figure 2-1. ModSecurity repository root



What you probably want is located in the `m2/` directory, which houses ModSecurity 2.x. Within that directory, you will find a directory structure that you will find familiar if you've worked with Subversion before:

```
m2/
    branches/
        2.1.x/
        2.5.x/
    experimental/
        [some stuff you won't care about]
    tags/
        [all official releases, each in own directory]
    trunk/
```

The `trunk` directory always contains the most recent development version. The active branches (currently only 2.5.x) may sometimes contain a feature or a fix that has not been released yet. The branches will always be generally stable anyway, and the risk of something breaking is minimal.

Once you have determined the location of the version of ModSecurity that you wish to use, you can get it using the export command of Subversion, like this:

```
$ svn export https://mod-security.svn.sourceforge.net/svnroot/mod-security/m2/trunk…
 modsecurity-trunk
```

What you will get in the folder `modsecurity-trunk` is almost the same as what you get when you download a release. The only practical difference will be that you won't have the documentation in PDF or HTML (although the DocBook sources will be there, as usual, in the `doc` subdirectory). The documentation build scripts are not in the main repository, but I don't think that will be a problem, because you can always read the sources directly (e.g., using the free XMLmind XML editor [http://www.xmlmind.com/xmleditor/]), or generate the missing files yourself (e.g., using the free XMLmind XSL-FO Converter [http://www.xmlmind.com/foconverter/]).

Compilation under Unix

Before you can start to compile ModSecurity, you must ensure that you have a complete development toolchain installed. Refer to the documentation of the operating system you are using for instructions. If you'll be adding ModSecurity to an operating system#provided Apache, you are likely to need to install a specific Apache development package, too. For example, on Debian and Ubuntu, you need to use `apache2-prefork-dev` or `apache2-threaded-dev`, depending on which deployment model (process-based or thread-based) you chose.

In the next step, ensure that you have resolved all the dependencies before compilation. The dependencies are listed in Table 2.2, "ModSecurity 2.5.x dependencies".

Table 2.2. ModSecurity 2.5.x dependencies

Dependency	In Apache?	Purpose
Apache Portable Runtime (`http://apr.apache.org`)	Yes	Various
APR-Util (`http://apr.apache.org`)	Yes	Various
`mod_unique_id`	Yes, but may not be installed by default	Generate unique transaction ID
libcurl (`http://curl.haxx.se/libcurl/`)	No	Remote logging using `mlogc`
libxml2 (`http://xmlsoft.org`)	No	XML processing
Lua 5.1+ (`http://www.lua.org`)	No	Writing complex rules in Lua (optional)
PCRE (`http://www.pcre.org`)	Yes, but cannot be used by ModSecurity	Regular expression matching

If you already have Apache installed, you will only ever need to deal with libcurl, libxml2, and Lua. With Apache compiled from source, you will also need the PCRE library. Although Apache comes bundled with it, it is used in a way that does not allow other modules to access it. To work around this issue, install PCRE separately and then tell Apache to use the external copy. I explain how to do that later in this section.

If you're installing from source, go to the packages' web sites, and download and install the tarballs. If you're using managed packages, you just need to determine what the missing packages are called. On Debian Lenny, the following command installs the missing packages:

```
# apt-get install libcurl3-dev liblua5.1-dev libxml2-dev
```

Refer to the documentation of the package management system used by your platform to discover how to search the package database.

The process should be straightforward from here on. Execute the following commands in succession:

```
$ ./configure
$ make
$ make mlogc
```

This set of commands ensures that you don't need any compile-time options. If you do, they are explained in the next section.

> **Note**
>
> Running additional tests after compilation (`make test` and `make test-regression`) is always a good idea, and is an especially good idea when using a development version of ModSecurity. If you are going to have any problems, you want to have them before installation, rather than after.

After ModSecurity is built, all you need to do is copy the two generated binaries to the binaries folder. For example:

```
$ sudo cp ./apache2/.libs/mod_security2.so /opt/modsecurity/bin
$ sudo cp ./tools/mlogc /opt/modsecurity/bin/
```

If you wish to keep the ModSecurity module along with the other Apache modules, just run the `install` target, which will produce the following output:

```
$ sudo make install
build/apxs-wrapper -i mod_security2.la
/usr/share/apache2/build/instdso.sh SH_LIBTOOL='/usr/share/apr-1.0/build/libtool'…
mod_security2.la /usr/lib/apache2/modules
/usr/share/apr-1.0/build/libtool --mode=install cp mod_security2.la…
/usr/lib/apache2/modules/
cp .libs/mod_security2.so /usr/lib/apache2/modules/mod_security2.so
cp .libs/mod_security2.lai /usr/lib/apache2/modules/mod_security2.la
PATH="$PATH:/sbin" ldconfig -n /usr/lib/apache2/modules
----------------------------------------------------------------------
Libraries have been installed in:
   /usr/lib/apache2/modules

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,--rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
----------------------------------------------------------------------
chmod 644 /usr/lib/apache2/modules/mod_security2.so
```

Installation does not activate or configure ModSecurity—you will need to do that manually, and I'll cover the process in the following sections.

Compile-time options

The configuration example from the previous section assumed that the dependencies were all installed as system libraries. It also assumed that the `configure` script will figure everything out on its own. It may or may not, but chances are good that you will occasionally need to do something different; this is where the compile-time options, listed in Table 2.3, "Compile-time options", come in handy.

Table 2.3. Compile-time options

Option	Description
`--with-apr`	Specify location of the Apache Portable Runtime library.
`--with-apu`	Specify location of the APR-Util library.
`--with-apxs`	Specify the location of Apache through the location of the `apxs` script.
`--with-curl`	Specify the location of libcurl.
`--with-libxml`	Specify the location of libxml. Some older versions used `--with-xml` instead.
`--with-pcre`	Specify the location of PCRE.

Using ModSecurity with a custom-compiled version of Apache is more work than it could have been, because of one slight problem. ModSecurity uses PCRE for pattern matching. PCRE is a very popular library and it is integrated into many operating systems. The problem is that Apache bundles PCRE and uses the bundled version by default when you compile it from source. Unless you take care to avoid conflicts, you'll probably end up with Apache using the bundled version of PCRE and ModSecurity using the one provided by the operating system. The solution to this problem is to build Apache to use the external PCRE version too, which requires just one change to the way you build Apache.

The configure option you need (remember, this is the configuration of Apache, not ModSecurity) is `--with-pcre`, and it is used like this:

```
$ ./configure --with-pcre=/usr/bin/pcre-config
```

After you compile and install Apache, you can confirm that the external PCRE library is used with ldd:

```
$ ldd ~/apache/bin/httpd  | grep pcre
        libpcre.so.3 => /usr/lib/libpcre.so.3 (0xb7d3f000)
```

You should get the same result when you compile ModSecurity:

```
$ ldd ~/apache/modules/mod_security2.so | grep pcre
        libpcre.so.3 => /usr/lib/libpcre.so.3 (0xb7f4c000)
```

Installation from Binaries

As previously discussed, using a binary version of ModSecurity is often the easiest option. It is also the fastest way to get ModSecurity up and running. The list of the available binary packages is available from http://www.modsecurity.org/download/. Traditionally, the best support has been available on Debian, FreeBSD, and Windows. At the time of this writing, you can get the latest version of ModSecurity in binary form on all these three platforms.

Fedora Core, CentOS, and Red Hat Enterprise Linux

If you are a Fedora Core user and you are running version 11 or later, you can install ModSecurity directly from the official distribution, using `yum`:

```
# yum install mod_security
```

On CentOS and Red Hat Enterprise Linux, you have two options. One is to use the packages from EPEL [http://fedoraproject.org/wiki/EPEL] (Extra Packages for Enterprise Linux), which is a volunteer effort and part of the Fedora community. The other option is to use the custom packages built by Jason Litka [http://www.jasonlitka.com/yum-repository/]. Either way, the available packages support CentOS/RHEL 4.x and 5.x, on the i386 and x86_64 architectures. The installation process is the same as for Fedora Core. At the time of this writing, only version 2.5.9 is available for download (with 2.5.11 being the most recent).

Debian and Ubuntu

Debian was the first distribution to include ModSecurity, but also the first distribution to kick it out due to a licensing issue. (In a nutshell, the issue boils down to incompatibility between GPL v2 and ASL v2.) The ModSecurity license was clarified in version 2.5.6 with the addition of an open source exception, and that opened the door for it to get back into Debian. Alberto Gonzalez Iniesta has been a long-time supporter of ModSecurity on Debian, packaging it from the very beginning. He supported ModSecurity in his own unofficial repository, and is now the official packager.

If you are running the current Debian testing (*squeeze*), then installing ModSecurity is as easy as typing:

```
# apt-get install libapache-mod-security
```

This single command will download the package and install it, then activate the module in the Apache configuration. If you are running the current Debian stable version (*lenny*),

you can get still get the binaries, but you will first need to configure your system to use the backports package repository [http://www.backports.org].

> **Note**
>
> Don't forget that Debian uses a special system to manage Apache modules and sites. To activate and deactivate modules, use `a2emod` and `a2dismod`, respectively. To manage Apache, use `apache2ctl`.

Installation on Windows

ModSecurity was ported to Windows early on, in 2003, and has run well on this platform ever since. Windows binary packages of ModSecurity are maintained by Steffen Land, who runs Apache Lounge [http://www.apachelounge.com], a community for those who run Apache on Windows. In addition to ModSecurity, Steffen maintains his version of Apache itself, as well as many third-party modules you might want to run on Windows. The ModSecurity binary packages are consistently up to date, so you will have little trouble if you want to run the latest version. The download includes ModSecurity (with embedded Lua 5.1) as well as the LibXML dynamic library. At this time, however, there is no support for remote logging with mlogc.

> **Note**
>
> Although it might be possible to run Steffen's ModSecurity binaries with a version of Apache produced elsewhere, you should really use only the packages from a single location that are intended to be used together. If you don't, you may encounter unusual behavior and web server crashes.

The installation is quite easy. First, download the package and copy the dynamic libraries into the `modules/` folder (of the Apache installation). Then, modify your Apache configuration to activate ModSecurity:

```
LoadModule security2_module modules/mod_security2.so
```

You will also need to activate `mod_unique_id`. If this module is not already active, there will be a commented-out line in your configuration already. You just need to find it and uncomment it. If it isn't there, just add the following:

```
LoadModule unique_id_module modules/mod_unique_id.so
```

Now you can proceed to configure ModSecurity, as explained in the next chapter.

3 Configuration

Now that you have ModSecurity compiled and ready to run, we can proceed to the configuration. This section, with its many subsections, goes through every part of ModSecurity configuration, explicitly configuring every little detail:

- Going through all the configuration directives will give you a better understanding of how ModSecurity works. Even if there are features that you don't need immediately, you will learn that they exist and you'll be able to take advantage of them when the need arises.
- By explicitly configuring every single feature, you will foolproof your configuration against potential incompatible changes to defaults in future versions of ModSecurity.

In accordance with its philosophy, ModSecurity won't do anything implicitly. It won't even activate unless you tell it to. There are three reasons for that:

1. By not doing anything implicitly, we ensure that ModSecurity does only what you tell it to. This, on the one hand, ensures that you need to understand a feature before you use it, and, on the other hand, makes it clear that your configuration is your responsibility.
2. It is impossible to design a default configuration that works in all circumstances. We can give you a framework within which you can work (as I am doing in this section), but you still need to shape your configuration according to your needs.
3. Security is not free. You pay for it by the increased consumption of RAM, CPU, or the possibility that you may block a legitimate request. Incorrect configuration may cause problems, so we need you to carefully think about what you're doing.

The remainder of this section explains the proposed default configuration for ModSecurity. You can get a good overview of the default configuration simply by examining the configuration directives supported by ModSecurity, which are listed in Table 3.1, "Main configuration directives" (with the exception of the logging directives, which are listed in several tables in the Chapter 4, *Logging*).

Table 3.1. Main configuration directives

Directive	Description
SecArgumentSeparator	Sets the application/x-www-form-urlencoded parameter separator
SecCookieFormat	Sets the cookie parser version
SecDataDir	Sets the folder for persistent storage
SecRequestBodyAccess	Controls request body buffering
SecRequestBodyInMemoryLimit	Sets the size of the per-request memory buffer
SecRequestBodyLimit	Sets the maximum request body size ModSecurity will accept
SecRequestBodyNoFilesLimit	Sets the maximum request body size, excluding uploaded files
SecResponseBodyAccess	Controls response body buffering
SecResponseBodyLimit	Specifies the response body buffering limit
SecResponseBodyLimitAction	Controls what happens once the response body limit is reached
SecResponseBodyMimeType	Specifies a list of response body MIME types to inspect
SecResponseBodyMimeTypesClear	Clears the list of response body MIME types
SecRuleEngine	Controls the operation of the rule engine
SecTmpDir	Sets the folder for temporary files

Folder Locations

Your first configuration task is to decide where on the filesystem to put the various bits and pieces that every ModSecurity installation consists of. Installation layout is often a matter of taste, so it is difficult for me to give you advice. Similarly, different choices may be appropriate in different circumstances. For example, if you are adding ModSecurity to a web server and you intend to use it only occasionally, you may not want to use an elaborate folder structure, in which case you'll probably put the ModSecurity folder underneath Apache's. When you're using ModSecurity as part of a dedicated reverse proxy installation, however, a well–thought out structure is something that will save you a lot of time in the long run.

I prefer to always use an elaborate folder layout, because I like things to be neat and tidy, and because the consistency helps me when I am managing multiple ModSecurity installations. I start by creating a dedicated folder for ModSecurity (`/opt/modsecurity`) with multiple subfolders underneath. The subfolders that are written to at runtime are all grouped (in `/opt/modsecurity/var`), which makes it easy to relocate them to a different filesystem using a symbolic link. I end up with the following structure:

Binaries
`/opt/modsecurity/bin`

Configuration files
: `/opt/modsecurity/etc`

Audit logs
: `/opt/modsecurity/var/audit`

Persistent data
: `/opt/modsecurity/var/data`

Logs
: `/opt/modsecurity/var/log`

Temporary files
: `/opt/modsecurity/var/tmp`

File uploads
: `/opt/modsecurity/var/upload`

Getting the permissions right may involve slightly more effort, depending on your circumstances. Most Apache installations bind to privileged ports (e.g., 80 and 443), which means that the web server must be started as `root`, and that further means that `root` must be the principal owner of the installation. Because it's not good practice to stay `root` at runtime, Apache will switch to a low-privilege account (we'll assume it's `apache`) as soon as it initializes. You'll find the proposed permissions in Table 3.2, "Folder permissions".

Table 3.2. Folder permissions

Location	Owner	Group	Permissions
/opt/modsecurity	root	apache	rwxr-x---
/opt/modsecurity/bin	root	root	rwxr-x---
/opt/modsecurity/etc	root	root	rwx------
/opt/modsecurity/var	root	apache	rwxr-x---
/opt/modsecurity/var/audit	apache	root	rwx------
/opt/modsecurity/var/data	apache	root	rwx------
/opt/modsecurity/var/log	root	root	rwx------
/opt/modsecurity/var/tmp	apache	apache	rwxr-x---
/opt/modsecurity/var/upload	apache	root	rwx------

> **Note**
>
> As an exception to the proposed layout, you may want to reuse Apache's log directory for ModSecurity logs. If you don't, you'll have the error log separate from the debug log (and the audit log if you choose to use the serial logging format). In a reverse proxy installation in particular, it makes great sense to keep everything integrated and easier to find. There may be other good reasons for breaking con-

> vention. For example, if you have more than one hard disk installed and you use the audit logging feature a lot, you may want to split the I/O operations acrosss the disks.

I've arrived at the desired permission layout through the following requirements:

1. As already discussed, it is `root` that owns everything by default, and we assign ownership to `apache` only where that is necessary.
2. In two cases (`/opt/modsecurity` and `/opt/modsecurity/var`), we need to allow `apache` to access a folder so that it can get to a subfolder; we do this by creating a group, also called `apache`, of which user `apache` is the only member.
3. One folder, `/opt/modsecurity/var/log`, stands out, because it is the only folder underneath `/opt/modsecurity/var` where `apache` is not allowed to write. That folder contains log files that are opened by Apache early on, while it is still running as `root`. On any Unix system, you *must* have only one account with write access to that folder, and it has to be the principal owner. In our case, that must be `root`. Doing otherwise would create a security hole, whereby the `apache` user would be able to obtain partial `root` privileges using symlink trickery. (Essentially, in place of a log file, the `apache` user creates a symlink to some other `root`-owned file on the system. When Apache starts it runs as `root` and opens for writing the system file that the `apache` user would otherwise be unable to touch. By submitting requests to Apache, one might be able to control exactly what is written to the log files. That can lead to system compromise.)
4. A careful observer will notice that I've allowed group folder access to `/opt/modsecurity/var/tmp` (which means that any member of the `apache` group is allowed to read the files in the folder) even though this folder is owned by `apache`, which already has full access. This is because you will sometimes want to allow ModSecurity to exchange information with a third user account—for example, if you want to scan uploaded files for viruses (usually done using ClamAV). To allow the third user account to access the files created by ModSecurity, you just need to make it a member of the `apache` group and relax the file permissions using the `SecUploadFileMode` directive.

Configuration Layout

If you have anything but a trivial setup, spreading configuration across several files is necessary in order to make maintenance easier. There are several ways to do that, and some have more to do with taste than anything else, but in this section I will describe an approach that is good enough to start with.

Whatever configuration design I use, there is usually one main entry point, typically named `modsecurity.conf`, which I use as a bridge between Apache and ModSecurity. In my bridge file, I refer to any other ModSecurity files I might have, such as those listed in Table 3.3, "Configuration files".

Table 3.3. Configuration files

Filename	Description
`main.conf`	Main configuration file
`rules-first.conf`	Rules that need to run first
`rules.conf`	Your principal rule file
`rules-last.conf`	Rules that need to run last

Somewhere in your Apache configuration, use the following line to include the ModSecurity configuration:

```
Include /opt/modsecurity/etc/modsecurity.conf
```

Your main configuration file (`modsecurity.conf`) may thus contain only the following lines:

```
<IfModule mod_security2.c>
Include /opt/modsecurity/etc/main.conf
Include /opt/modsecurity/etc/rules-first.conf
Include /opt/modsecurity/etc/rules.conf
Include /opt/modsecurity/etc/rules-last.conf
</IfModule>
```

The `<IfModule>` tag is there to ensure that the ModSecurity configuration files are used only if ModSecurity is active in the web server. This is common practice when configuring any nonessential Apache modules and allows you to deactivate a module simply by commenting out the appropriate `LoadModule` line.

Adding ModSecurity to Apache

As the first step, make Apache aware of ModSecurity, adding the needed components. Depending on how you've chosen to run ModSecurity, this may translate to adding one or more lines to your configuration file. This is what the lines may look like:

```
# Load libxml2
LoadFile /usr/lib/libxml2.so
# Load Lua
```

```
LoadFile /usr/lib/liblua5.1.so
# Finally, load ModSecurity
LoadModule security2_module modules/mod_security2.so
```

Now you just need to tell Apache where to find the configuration:

```
Include /opt/modsecurity/etc/modsecurity.conf
```

Powering Up

ModSecurity has a master switch—the `SecRuleEngine` directive—that allows you to quickly turn it on and off. This directive will always come first in every configuration. I generally recommend that you start in detection-only mode, because that way you are sure nothing will be blocked.

```
# Enable ModSecurity, attaching it to every transaction.
SecRuleEngine DetectionOnly
```

You will normally want to keep this setting enabled, of course, but there will be cases in which you won't be exactly sure whether ModSecurity is doing something it shouldn't be. Whenever that happens, you will want to set it to `Off`, just for a moment or two, until you perform a request without it running.

The `SecRuleEngine` directive is context-sensitive (i.e., it works with Apache's container tags `<VirtualHost>`, `<Location>`, and so on), which means that you are able control exactly where ModSecurity runs. You can use this feature to enable ModSecurity only for some site, parts of a web site, or even for a single script only. I discuss this feature in detail later.

Will ModSecurity Block in Detection-Only Mode?

You may expect ModSecurity never to block when you configure `SecRuleEngine` with `DetectionOnly`, but that's not actually the case. There are two edge cases in which ModSecurity will block, to the surprise of most users. In both cases, an error occurs while ModSecurity is handling inbound or outbound data. If ModSecurity sees more data than it was configured to handle, it will respond with a `HTTP_REQUEST_ENTITY_TOO_LARGE` (417) error code. The three directives to watch are: `SecRequestLimit`, `SecRequestNoFilesLimit`, and `SecResponseBodyLimit`.

Problems with response body limits only can be worked around by changing the `SecResponseBodyLimitAction` setting to `ProcessPartial`. For the other two directives, we will have to wait for a future version of ModSecurity. I submitted a proposal to fix this behavior (issue `MODSEC-104`), and I expect that we will see a true detection-only mode in ModSecurity 2.6.

Request Body Handling

Requests consist of two parts: the headers part, which is always present, and the body, which is optional. Use the `SecRequestBodyAccess` directive to tell ModSecurity to look at request bodies:

```
# Allow ModSecurity to access request bodies. If you don't,
# ModSecurity won't be able to see any POST parameters
# and that's generally not what you want.
SecRequestBodyAccess On
```

Once this feature is enabled, ModSecurity will not only have access to the content transmitted in request bodies, but it will also completely buffer them. The buffering is essential for reliable attack prevention. With buffering in place, your rules have the opportunity to inspect requests in their entirety, and only after you choose not to block will the requests be allowed through.

The downside of buffering is that, in most cases, it uses RAM for storage, which needs to be taken into account when ModSecurity is running embedded in a web server. There are three directives that control how buffering is done. The first two, `SecRequestBodyLimit` and `SecRequestBodyNoFilesLimit`, establish request limits:

```
# Maximum request body size we will accept for buffering.
# If you support file uploads then the value given on the
# first line has to be as large as the largest file you
# want to accept. The second value refers to the size of
# data, with files excluded. You want to keep that value
# as low as practical.
SecRequestBodyLimit 1310720
SecRequestBodyNoFilesLimit 131072
```

In the versions prior to 2.5, ModSecurity supported only `SecRequestBodyLimit` (which establishes an absolute limit on a request body), but that directive turned out to be impractical in combination with file uploads. File uploads generally do not use RAM (and thus do not create an opportunity for a denial of service attack), which means that it is safe to allow large such requests. Unfortunately, doing so also meant allowing large requests that are not file uploads, defying the purpose for which the directive was introduced in the first place. The second directive, `SecRequestBodyNoFilesLimit`, which was introduced in ModSecurity 2.5, calculates request body sizes slightly differently, ignoring the space taken up by files. In practice, this means that the maximum allowed request body size will be that specified in the `SecRequestBodyNoFilesLimit` directive, with the exception of file uploads, where the setting in `SecRequestBodyLimit` takes precedence.

> **Note**
>
> ModSecurity will respond with a 413 (Request Entity Too Large) response status code when a request body limit is reached. This response code was chosen to mimic what Apache would have done in similar circumstances.

The third directive that deals with buffering, `SecRequestBodyInMemoryLimit`, controls how much of a request body will be stored in RAM, but it only works with file upload (`multipart/form-data`) requests:

```
# Store up to 128 KB of request body data in memory. When
# the multipart parser reaches this limit, it will start
# using your hard disk for storage. That is generally slow,
# but unavoidable.
SecRequestBodyInMemoryLimit 131072
```

The request bodies that fit within the limit configured with `SecRequestBodyInMemoryLimit` will be stored in RAM. The request bodies that are larger will be streamed to disk. This directive allows you to trade performance (storing request bodies in RAM is fast) for size (the storage capacity of your hard disk is much bigger than that of your RAM).

Response Body Handling

Similarly to requests, responses consist of headers and a body. Unlike requests, however, most responses have bodies. Use the `SecResponseBodyAccess` directive to tell ModSecurity to observe (and buffer) response bodies:

```
# Allow ModSecurity to access response bodies. We leave
# this disabled because most deployments want to focus on
# the incoming threats, and leaving this off reduces
# memory consumption.
SecResponseBodyAccess Off
```

I prefer to start with this setting disabled, because many deployments don't care to look at what leaves their web servers. Keeping this feature disabled means ModSecurity will use less RAM and less CPU. If you care about output, however, just change the directive setting to `On`.

There is a complication with response bodies, because you generally only want to look at the bodies of some of the responses. Response bodies make the bulk of the traffic on most web sites, and the majority of that are just static files that don't have any security relevance in most cases. The response MIME type is used to distinguish the interesting responses from the ones that are not. The `SecResponseBodyMimeType` directive lists the response MIME types you are interested in.

```
# Which response MIME types do you want to look at? You
# should adjust the configuration below to catch documents
# but avoid static files (e.g., images and archives).
SecResponseBodyMimeType text/plain text/html
```

> **Note**
>
> To instruct ModSecurity to inspect the response bodies for which the MIME type is unknown (meaning that it was not specified in the response headers), use the special string (`null`) as a parameter to `SecResponseBodyMimeType`.

You can control the size of a response body buffer using the `SecResponseBodyLimit` directive:

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

The problem with limiting the size of a response body buffer is that it breaks sites whose pages are longer than the limit. In ModSecurity 2.5, we introduced the `SecResponseBodyLimitAction` directive, which allows ModSecurity users to choose what happens when the limit is reached:

```
# What happens when we encounter a response body larger
# than the configured limit? By default, we process what
# we have and let the rest through.
SecResponseBodyLimitAction ProcessPartial
```

If the setting is `Reject`, the response will be discarded and the transaction interrupted with a 500 (Internal Server Error) response code. If the setting is `ProcessPartial`, which I recommend, ModSecurity will process what it has in the buffer and allow the rest through.

At the first thought, it may seem that allowing the processing of partial response bodies creates a security issue. For the attacker who controls output, it seems easy to create a response that is long enough to bypass observation by ModSecurity. This is true. However, if you have an attacker with full control of output, it is impossible for any type of monitoring to work reliably. For example, such an attacker could encrypt output, in which case it will be opaque to us. Response body monitoring works best to detect information leakage, configuration errors, traces of attacks (successful or not), and data leakage in the cases when an attacker does not have full control of output.

Other than that, response monitoring is most useful when it comes to preventing the data leakage that comes from low-level error messages (e.g., database problems). Because such messages typically appear near the beginning of a page, the `ProcessPartial` setting will work just as well to catch them.

> **Note**
>
> If you are using `mod_deflate` for response compression, it may, under some circumstances, cause ModSecurity to assess and log compressed data (which is not very useful, and may even cause false positives). It is not clear what causes the problem, but a fix to deal with the problem is in ModSecurity 2.5.11 and later.

Filesystem Locations

We've made the decisions regarding filesystem locations already, so all we need to do now is translate them into configuration. The following two directives tell ModSecurity where to create temporary files (`SecTmpDir`) and where to store persistent data (`SecDataDir`):

```
# The location where ModSecurity will store temporary files
# (for example, when it needs to handle a multipart request
# body that is larger than the configured limit). If you don't
# specify a location here your system's default will be used.
# It is recommended that you specify a location that's private.
SecTmpDir /opt/modsecurity/var/tmp/

# The location where ModSecurity will keep its data. This,
# too, needs to be a path that other users can't access.
SecDataDir /opt/modsecurity/var/data/
```

File Uploads

Next, we configure the handling of file uploads. We configure the folder where ModSecurity will store intercepted files, but we keep this functionality disabled for now. File upload interception slows down ModSecurity and can potentially consume a lot of disk space, so you'll want to enable this functionality only in the places where you really need it.

```
# The location where ModSecurity will store intercepted
# uploaded files. This location must be private to ModSecurity.
SecUploadDir /opt/modsecurity/var/upload/

# By default, do not intercept (nor store) uploaded files.
SecUploadKeepFiles Off
```

For now, we also assume that you will not using be external scripts to inspect uploaded files. That allows us to keep the file permissions more secure, by allowing only the `apache` user access:

```
# Uploaded files are by default created with permissions that
# do not allow any other user to access them. You may need to
```

```
# relax that if you want to interface ModSecurity to an
# external program (e.g., an anti-virus).
SecUploadFileMode 0600
```

Debug Log

Debug logging is very useful for troubleshooting, but in production you want to keep it at minimum, because too much logging will affect the performance. The recommended debug log level for production is 3, which will duplicate in the debug log what you will also see in Apache's error log. This is handy, because the error log will grow at a faster rate and may be rotated. A copy of the ModSecurity messages in the debug log means that you always have all the data you need.

```
# Debug log
SecDebugLog /opt/modsecurity/var/log/debug.log
SecDebugLogLevel 3
```

Audit Log

In ModSecurity terminology, "audit logging" refers to logging complete transaction data. For a typical transaction without a request body, this translates to roughly 1 KB. Multiply that by the number of requests you are receiving daily and you'll soon realize that you want to keep this type of logging to an absolute minimum.

Our default configuration will use audit logging only for the transactions that are *relevant*, which means those that have had an error or a warning reported against them. Other possible values for `SecAuditEngine` are `On` (log everything) and `Off` (log nothing).

```
# Log only what is really necessary.
SecAuditEngine RelevantOnly
```

In addition, we will also log the transactions with response status codes that indicate a server error (500–599). You should never see such transactions on an error-free server. The extra data logged by ModSecurity may help you uncover security issues, or problems of some other type.

```
# Also log requests that cause a server error.
SecAuditLogRelevantStatus ^5
```

By default, we log all transaction data except response bodies. This assumes you will seldom log (as it should be), because response bodies can take up a lot of space.

```
# Log everything we know about a transaction.
```

```
SecAuditLogParts ABCDEFHKZ
```

Using the same assumption, we choose to use a single file to store all the recorded information. This is not adequate for the installations that will log a lot and prevents remote logging, but it is good enough to start with:

```
# Use a single file for logging.
SecAuditLogType Serial
SecAuditLog /opt/modsecurity/var/log/audit.log
```

As the final step, we will configure the path that will be used in the more scalable audit logging scheme, called *concurrent logging*, even though you won't need to use it just yet:

```
# Specify the path for concurrent audit logging.
SecAuditLogStorageDir /opt/modsecurity/var/audit/
```

Miscellaneous Options

The directives covered in this section are seldom needed, but having them will allow us to achieve complete coverage of the ModSecurity configuration options. You'll also be aware that they exist and will be able to use them in the rare cases where they are needed.

The `SecArgumentSeparator` directive allows you to change the parameter separator used for the `application/x-www-form-urlencoded` encoding, which is used to transport all `GET` parameters and most `POST` parameters.

```
SecArgumentSeparator &
```

Virtually all applications use an ampersand for this purpose, but some may not. The HTML 4.01 specification recommends that applications support the use of semicolons as separators (see section *B.2.2 Ampersands in URI attribute values*) for convenience. In PHP, for example, it is possible to use any character as a separator.

The `SecCookieFormat` directive selects one of the two cookie parsers available in ModSecurity. Virtually all applications use Netscape-style cookies (sometimes also known as version 0) cookies, so there will be little reason to change this setting:

```
SecCookieFormat 0
```

Default Rule Match Policy

As we're nearing the end of the configuration, you need to decide what you want to happen when a rule matches. It is recommended that you start with only detection (and not block-

ing), because that will allow you to monitor the operation of your installation over a period of time and ensure that legitimate traffic is not being marked as suspicious:

```
SecDefaultAction "phase:1,log,auditlog,pass"
```

This default policy will work for all rules that follow it in the same configuration context. (For more information, turn to the section called "Configuration contexts".)

> **Note**
>
> It is possible to write rules that ignore the default policies. If you are using third-party rule sets and you are not sure how they will behave, consider switching the entire engine to detection only (using `SecRuleEngine`). No rule will block when you do that, regardless of how it was designed to work.

Handling Parsing Errors

As you may recall from our earlier discussion, ModSecurity avoids making decisions for you. It will detect problems as they occur, but it will generally leave to you to deal with them. In our default configuration, we will have two rules to deal with the situations that ModSecurity can't deal with it on its own—parser failures.

ModSecurity parsers are designed to be as permissive as possible without compromising security. Even when they do not fail, they may raise warning flags for you to detect in rules. But no matter how permissive you are, you will encounter traffic that is so malformed that you don't know how to handle it.

The first rule will examine the `REQBODY_PROCESSOR_ERROR` flag for errors. This flag will be raised whenever a request body parsing error occurs, regardless of which parser was used for parsing:

```
# Verify that we've correctly processed the request body.
# As a rule of thumb, when failing to process a request body
# you should reject the request (when deployed in blocking mode)
# or log a high-severity alert (when deployed in detection-only mode).
SecRule REQBODY_PROCESSOR_ERROR "!@eq 0" \
    "phase:2,t:none,log,block,msg:'Failed to parse request body: …
%{REQBODY_PROCESSOR_ERROR_MSG}'"
```

The second rule is specific to the `multipart/form-data` parser, which is used to handle file uploads. If it detects a problem, it produces an error message detailing the flaws:

```
# By default be strict with what we accept in the multipart/form-data
# request body. If the rule below proves to be too strict for your
# environment consider changing it to detection-only. You are encouraged
```

```
# _not_ to remove it altogether.
SecRule MULTIPART_STRICT_ERROR "!@eq 0" \
"phase:2,t:none,log,block,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}'"
```

Errors specific to the multipart parsers should never occur unless an attacker genuinely tries to bypass ModSecurity by manipulating the request body payload. Some versions of ModSecurity did have false positives in this area, but the most recent version (2.5.9 at the time of writing) should be false-positive-free. If you do encounter such a problem, feel free to post it to the mod-security-users mailing list. It will mean that you've encountered an interesting attacker or a ModSecurity bug.

Verifying Installation

After you're done installing and configuring ModSecurity, it is recommended that you undertake a short exercise to ensure everything is in order:

1. Add a simple blocking rule to detect something in a parameter. For example, the following rule will inspect all parameters for the word `script`, responding with a 503 (*Service Unavailable*) on a match:

```
SecRule ARGS MY_UNIQUE_TEST_STRING \
    "phase:1,log,deny,status:503"
```

2. Restart Apache, using the `graceful` restart option if your server is in production and you don't want any downtime.
3. Send a `GET` request, using your browser, to the ModSecurity-protected server, including the "attack payload" in a parameter (i.e., `http://www.example.com/?test=MY_UNIQUE_TEST_STRING`). ModSecurity should block the request.
4. Verify that the message has appeared in both the error log and the debug log, and that the audit log contains the complete transaction.
5. Submit a `POST` request that triggers the test rule. With this request, you are testing whether ModSecurity will see the request body, and whether it will be able to pass the data in it to your backend after inspection. For this test, in particular, it is important that you're testing with the actual application you want to protect. Only doing

so will exercise the entire stack of components that make the application. This test is important because of the way Apache modules are written (very little documentation, so module authors generally employ any approach that "works" for them)—you can generally never be 100% certain that a third-party module was implemented correctly. For example, it is possible to write a module that will essentially hijack a request early on and bypass all the other modules, including ModSecurity. We are doing this test simply because we don't want to leave anything to chance.

6. If you want to be really pedantic (I have been, on many occasion—you can never be too sure), you may want to consider writing a special test script for your application, which will somehow record the fact that it has been invoked (mine usually writes to a file in `/tmp`). By sending a request that includes an attack—which will be intercepted by ModSecurity—and verifying that the script has not been invoked, you can be completely sure that blocking works as intended.
7. Remove the test rule and restart Apache again.
8. Finally, and just to be absolutely sure, examine the permissions on all Apache and ModSecurity locations and verify that they are correct.

You're done!

4 Logging

This section covers the logging capabilities of ModSecurity in detail. Logging is a big part of what ModSecurity does, so it will not surprise you to learn that there are extensive facilities available for your use.

Debug Log

The debug log is going to be your primary troubleshooting tool, especially initially, while you're learning how ModSecurity works. You are likely to spend a lot of time with the debug log cranked up to level 9, observing why certain things work the way they do. There are two debug log directives, as you can see in Table 4.1, "Debug log directives".

Table 4.1. Debug log directives

Directive	Description
`SecDebugLog`	Path to the debug log file
`SecDebugLogLevel`	Debug log level

In theory, there are 10 debug log levels, but not all are used. You'll find the ones that are in Table 4.2, "Debug log levels". Messages with levels 1–3 are designed to be meaningful, and are copied to the Apache's error log. The higher-level messages are there mostly for troubleshooting and debugging.

Table 4.2. Debug log levels

Debug log level	Description
0	No logging
1	Errors (e.g., fatal processing errors, blocked transactions)
2	Warnings (e.g., non-blocking rule matches)
3	Notices (e.g., non-fatal processing errors)
4	Informational
5	Detailed
9	Everything!

You will want to keep the debug log level in production low (either at 3 if you want a copy of all messages in the debug log, or at 0 if you're happy with having them only in the error log). This is because you can expect in excess of 50 debug log messages (each message is an I/O operation) and at least 7 KB of data for an average transaction. Logging all that for every single transaction consumes a lot of resources.

This is what a single debug log line looks like:

```
[18/Aug/2009:08:18:08 +0100] [192.168.3.111/sid#80f4e40][rid#81d0588][/index.html]…
[4] Initialising transaction (txid SopVsH8AAAEAAE8-NB4AAAAD).
```

The line starts with metadata that is often longer than the message itself: the time, client's IP address, internal server ID, internal request ID, request URI, and finally, the debug log level. The rest of the line is occupied by the message, which is essentially free-form. You will find many examples of debug log messages throughout this guide, which I've used to document how ModSecurity works.

Debugging in Production

There's another reason for avoiding extensive debug logging in production, and that's simply that it's very difficult. There's usually so much data that it sometimes takes you ages to find the messages pertaining to the transaction you wish to investigate. In spite of the difficulties, you may occasionally need to debug in production because you can't reproduce a problem elsewhere.

> **Note**
>
> ModSecurity 2.5 extended the audit logging functionality by being able to record in the audit log all the rules that matched. This feature is very helpful, as it minimizes the need for debugging in production, but it still can't tell you why some rules *didn't* match.

One way to make debugging easier is to keep debug logging disabled by default and enable it only for the part of the site you wish to debug. You can do this by overriding the default configuration using the `<Location>` context directive. While you're doing that, it may be a good idea to specify a different debug log file altogether. That way you'll keep main debug log file free of your tests.

```
<Location /myapp/>
    SecDebugLogLevel 9
    SecDebugLog /opt/modsecurity/var/log/troubleshooting.log
</Location>
```

This approach, although handy, still does not guarantee that the volume of information in the debug log will be manageable. What you really want is to enable debug logging for the requests you send. ModSecurity provides a solution for this by allowing a debug log level to be changed at runtime, on a per-request basis. This is done using the special `ctl` action that allows some of the configuration to be updated at runtime.

All you need to do is somehow uniquely identify yourself. In some circumstances, observing the IP address will be sufficient:

```
SecRule REMOTE_ADDR "@streq 192.168.1.1" \
    phase:1,nolog,pass,ctl:debugLogLevel=9
```

Using your IP address won't work in the cases when you're hidden by a NAT of some sort, and share an IP address with a bunch of other users. One straightforward approach is to modify your browser settings to put a unique identifier in your User-Agent request header. (How exactly that is done depends on the browser you are using. In Firefox, for example, you can add a general.useragent.override setting to your configuration, or use one of the many extensions specifically designed for this purpose.)

```
SecRule REQUEST_COOKIES:User-Agent YOUR_UNIQUE_ID \
    phase:1,nolog,pass,ctl:debugLogLevel=9
```

This approach, although easy, has a drawback: all your requests will cause an increase in debug logging. You may think of an application in terms of dynamic pages, but extensive debug logging will be enabled for every single embedded object, too. Also, if you're dealing with an application that you're using frequently, you may want to avoid excessive logging.

The most accurate way of dynamically enabling detailed debug logging is to manually indicate, to ModSecurity, the exact requests on which you want it to increase logging. You can do this by modifying your User-Agent string on request-by-request basis, using one of the tools that support request interception and modification. (The Tamper Data extension does that for Firefox.) Armed with such a tool, you submit your requests in your browser, trap it in request, modify it, and then allow it through. It's a bit involved, but a time-saver overall.

And, while you are at it, it is a good idea to make your identifiers similar enough for your rule to always detect them, but different enough to allow you to use a search function to quickly find the exact request in a file with thousands.

Audit Log

It is a little-known fact that I originally started to work on ModSecurity because I was frustrated with not being able to log full HTTP transaction data. The audit log, which does just that, was one of the first features implemented.

Table 4.3. Audit log directives

Directive	Description
`SecAuditEngine`	Controls the audit log engine; possible values `On`, `Off`, or `RelevantOnly`
`SecAuditLog`	Path to an audit log file
`SecAuditLog2`	Path to another audit log file (copy)
`SecAuditLogParts`	Specifies which part of a transaction will be logged
`SecAuditLogRelevantStatus`	Specifies which response statuses will be considered relevant
`SecAuditLogStorageDir`	Path there concurrent audit log files will be stored
`SecAuditLogType`	Specifies the type of audit log to use: `Serial` or `Concurrent`

A typical audit log entry (short, GET request without a body and no logging of the response body) consumes around 1.3 KB. Requests with bodies will increase the amount of data that needs to be logged, as well as the logging of response bodies.

Logically, each audit log entry is a single file. When serial audit logging is used, all entries will be placed within one file, but with concurrent audit logging, one file per entry is used. Looking at a single audit log entry, you'll find that it consists of multiple independent segments (parts):

```
--6b253045-A--
...
--6b253045-B--
...
--6b253045-C--
...
--6b253045-F--
...
--6b253045-E--
...
--6b253045-H--
...
```

```
--6b253045-Z--
```

A segment begins with a boundary and ends when the next segment begins. The only exception is the terminating segment (`Z`), which consists only of the boundary. The idea behind the use of multiple segments is to allow each audit log entry to contain potentially different information. Only the parts `A` and `Z` are mandatory; the use of the other parts is controlled with the `SecAuditLogParts` directive. Table 4.4, "Audit log parts"contains the list of all audit log parts, along with a description of their purpose.

Table 4.4. Audit log parts

Part letter	Description
A	Audit log header (mandatory)
B	Request headers
C	Request body
D	Reserved
E	Response body
F	Response headers
G	Reserved
H	Audit log trailer, which contains additional data
I	Compact request body alternative (to part C), which excludes files
J	Reserved
K	Contains a list of all rules that matched for the transaction
Z	Final boundary (mandatory)

Audit Log Entry Example

Every audit log entry begins with part `A`, which contains the basic information about the transaction: time, unique ID, source IP address, source port, destination IP address, and destination port:

```
--6b253045-A--
[18/Aug/2009:08:25:15 +0100] SopXW38EAAE9YbLQ 192.168.3.1 2387 192.168.3.111 8080
```

Part `B` contains the request headers and nothing else:

```
--6b253045-B--
POST /index.html?a=test HTTP/1.1
Host: 192.168.3.111:8080
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
```

```
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://192.168.3.111:8080/index.html?a=test
Content-Type: application/x-www-form-urlencoded
Content-Length: 6
```

Part C contains the raw request body, typically that of a POST request:

```
--6b253045-C--
b=test
```

Part F contains the response headers:

```
--6b253045-F--
HTTP/1.1 200 OK
Last-Modified: Tue, 18 Aug 2009 07:17:44 GMT
ETag: "6eccf-99-4716550995f20"
Accept-Ranges: bytes
Content-Length: 159
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
```

Part E contains the response body:

```
--6b253045-E--
<html><body><h1>It works!</h1></body></html>

<form action="index.html?a=test" method="POST">
<textarea name="b">test</textarea>
<input type=submit>
</form>
```

The final part, H, contains additional transaction information.

```
--6b253045-H--
Stopwatch: 1250580315933960 1902 (551* 580 978)
Response-Body-Transformed: Dechunked
Producer: ModSecurity for Apache/2.5.9 (http://www.modsecurity.org/).
Server: Apache/2.2.11 (Unix) DAV/2
```

Part K contains a list of rules that matched in a transaction. It is not unusual for this part to be empty, but if you have a complex rule set, it may show quite a few rules. Audit logs that record transactions on which there were warnings, or those that were blocked, will contain at least one rule here. In this example, you'll find a rule that emits a warning on every request:

```
--6b253045-K--
SecAction "phase:2,auditlog,log,pass,msg:'Matching test'"
```

Every audit log file ends with the terminating boundary, which is part `Z`:

```
--6b253045-Z--
```

Concurrent Audit Log

Initially, ModSecurity supported only the serial audit logging format. Concurrent logging was introduced to address two issues:

- Serial logging is only adequate for moderate use, because only one audit log entry can be written at any one time. Serial logging is fast (logs are written at the end of every transaction, all in one go) but it does not scale well. In the extreme, a web server performing full transaction logging practically processes only one request at any ont time.
- Real-time audit log centralization requires a manipulation of one audit log entry at a time (it needs to be read, transported to a remote server and then deleted), making storage within only one file very impractical.

Concurrent audit logging changes the operation of ModSecurity in two aspects. To observe the changes, switch to concurrent logging without activating `mlogc by` changing `SecAuditLogType` to `Concurrent` (don't forget to restart Apache).

First, as expected, each audit log entry will be stored in a separate file. But the files will not be created directly in the folder specified by `SecAuditLogStorageDir`, as you might have expected. Instead, an elaborate two-levels-deep structure of subfolders (with names based on time) will be created underneath, with only the audit log entries created within the same minute sharing folders:

```
dev:/opt/modsecurity/var/audit# find .
.
./20090822
./20090822/20090822-1324
./20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQAAAAAA
./20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQEAAAAA
```

The purpose of the scheme is to prevent too many files from being created within one directory; many filesystems have limits that can be relatively quickly reached on a busy web server. The first two parts in each filename are based on time (`YYYYMMDD` and `HHMMSS`). The third parameter is the unique transaction ID.

In addition to each entry getting its own file, the format of the main audit log file will change when concurrent logging is activated. The file that previously stored the entries themselves

will now be used as a record of all generated audit log files. It is sometimes called an index file.

```
192.168.3.130 192.168.3.1 - - [22/Aug/2009:13:24:20 +0100] "GET / HTTP/1.1" 200…
56 "-" "-" SojdH8AAQEAAAugAQAAAAAA "-"…
/20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQAAAAAA 0 1248…
md5:8b097f4f880852e179e7b63b68a7fc92
192.168.3.130 192.168.3.1 - - [22/Aug/2009:13:24:20 +0100] "GET /favicon.ico…
HTTP/1.1" 404 267 "-" "-" SojdH8AAQEAAAugAQEAAAAA "-"…
/20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQEAAAAA 0 1226…
md5:c76740f076a3cb759d62fb610ab39342
```

The index file is similar in principle to a web server access log. Each line describes one transaction, duplicating some of the information already available in audit log entries. The purpose of the index file is two-fold:

- The first part, which duplicates some of the information available in audit logs, serves as a record of everything that you have recorded so that you can easily search through it.
- The second part tells you where an audit log entry is stored (e.g., `/20090822/20090822-1324/20090822-132420-SojdH8AAQEAAAugAQEAAAAA`), where it begins within that file (always zero, because this feature is not used), how long it is, and gives you its MD5 hash (useful to verify integrity).

When real-time audit log centralisation is used, this information is not written to a file. Instead, it is written to a pipe, which means that it is sent directly to another process, which deals with the information immediately. You will see how that works in the next section.

Remote Logging

ModSecurity comes with a tool called `mlogc` (short for ModSecurity Log Collector), which can be used to transport audit logs in real time to a remote logging server. This tool has the following characteristics:

Secure
: The communication path between your ModSecurity sensors and the remote logging server is secured using SSL. Mlogc will authenticate itself to the server using Basic Authentication.

Efficient
: Mlogc is a robust multithreaded program that submits multiple audit log entries in parallel, reusing the existing connections wherever possible.

Reliable
: An audit log entry will be deleted from the sensor only once its safe receipt is acknowledged by the logging server.

Buffered
: Mlogc maintains its own audit entry queue, which has two benefits. First, if the logging server is not available the entries will be preserved, and submitted once the servers comes back online. Second, mlogc controls the rate at which audit log entries are submitted, meaning that a burst of activity on a sensor will not result in an uncontrolled burst of activity on the remote logging server.

If you've followed my installation instructions, you will have mlogc compiled and sitting in your `bin/` folder. To proceed, you will need to configure it, then add it to the ModSecurity configuration. Mlogc uses a simple yet effective protocol based on HTTP. The protocol is documented in the *ModSecurity 2 Data Formats* document, which is distributed with ModSecurity starting with version 2.5.10.

How Mlogc Works

Remote logging in ModSecurity is implemented through an elaborate scheme designed to minimize the possibility of data loss. Here is how it's done:

1. ModSecurity processes a transaction and creates an audit log entry file on disk, as explained in the the section called "Concurrent Audit Log".
2. ModSecurity then notifies the mlogc tool, which runs in parallel, in a separate process. The notification contains the enough information to locate the audit log entry file on disk.
3. Mlogc adds the audit log entry information to the in-memory queue, but also to the transaction log (usually `mlogc-transaction.log`).
4. Mlogc operates a number of worker processes, which will each take one audit log entry at a time and submit them to a remote logging server. After an entry is successfully submitted, it is removed from the in-memory queue and recorded in the transaction log.
5. Mlogc will periodically pause to perform a checkpoint operation: it will write the in-memory queue to the disk (usually `mlogc-queue.log`) and erase the transaction log.

If mlogc crashes, or something else unexpected happens with the server, mlogc will detect an unclean shutdown on the next invocation. It will then reconstruct the entry queue using the last known good point (the on-disk queue) and the record of events since the moment the on-disk queue was created, in the transaction log.

Configuring Mlogc

The mlogc configuration file is similar to that of Apache, only simpler. First we need to tell mlogc where its "home" is, which is where it will create its log files. Log files are very important, because—as it is Apache that starts mlogc and ModSecurity that talks to it—we never interact with mlogc directly. We'll need to look in the log files for clues in case of problems.

```
# Specify the folder where the logs will be created
CollectorRoot  /opt/modsecurity/var/log

# Define what the log files will be called. You probably
# won't ever change the names, but mlogc requires you
# to define it.
ErrorLog       mlogc-error.log

# The error log level is a number between 0 and 5, with
# level 3 recommended for production (5 for troubleshooting).
ErrorLogLevel 3

# Specify the names of the data files. Similar comment as
# above: you won't want to change these, but they are required.
TransactionLog mlogc-transaction.log
QueuePath      mlogc-queue.log
LockFile       mlogc.lck
```

Then we tell it where to find audit log entries. The value given to `LogStorageDir` should be the same as you provided to ModSecurity's `SecAuditLogStorageDir`:

```
# Where are the audit log entries created by ModSecurity?
LogStorageDir /opt/modsecurity/var/data
```

Next comes the information where the audit log entires are to be submitted. We identify a remote server with an URL and credentials:

```
# Remote logging server details.
ConsoleURI "https://REMOTE_ADDRESS:8888/rpc/auditLogReceiver"
SensorUsername "USERNAME"
SensorPassword "PASSWORD"
```

The remaining configuration directives aren't required, but it's always good idea to explicitly configure your programs rather than let them use their defaults:

```
# How many parallel connections to use to talk to the server,
# and how much to wait (in milliseconds) between submissions.
# These two directives are used to control the rate at which
# audit log entries are submitted.
```

```
MaxConnections      10
TransactionDelay    50

# How many entries is a single thread allowed to process
# before it must shut down.
MaxWorkerRequests   1000

# How long to wait at startup before really starting.
StartupDelay        5000

# Checkpoints are periods when the entries from the transaction
# log (which is written to sequentially) are consolidated with
# the entries in the main queue.
CheckpointInterval  15

# Back-off time after goes away or responds with a fatal error.
ServerErrorTimeout  60
```

> **Note**
>
> Mlogc will take audit log entries created by ModSecurity, submit them to a remote logging server and delete them from disk, but it will leave the empty folders (that were used to store the entries) behind. You will have to remove them yourself, either manually or with a script.

Activating Mlogc

You will need to make two changes to your default configuration. First you need to switch to concurrent audit logging, because that's the only way mlogc can work:

```
SecAuditLogType Concurrent
```

Next you need to activate `mlogc`, which is done using the piped-logging feature of Apache:

```
SecAuditLog "|/opt/modsecurity/bin/mlogc /opt/modsecurity/etc/mlogc.conf"
```

The pipe character at the beginning of the line tells Apache to treat what follows as a command line. As a result, whenever you start Apache from now on, it will start a copy of `mlogc` in turn, and keep it running in parallel, leaving a one-way communication channel that will be used by ModSecurity to inform mlogc of every new audit log entry it creates.

Your complete configuration should look like this now:

```
SecAuditEngine RelevantOnly
SecAuditLogRelevantStatus ^5
SecAuditLogParts ABCDEFHKZ
SecAuditLogType Concurrent
```

```
SecAuditLog "|/opt/modsecurity/bin/mlogc /opt/modsecurity/etc/mlogc.conf"
SecAuditLogStorageDir /opt/modsecurity/var/audit/
```

If you restart Apache now, you should see `mlogc` running:

```
USER       PID  COMMAND
root     11845  /usr/sbin/apache2 -k start
root     11846  /opt/modsecurity/bin/mlogc /opt/modsecurity/etc/mlogc.conf
apache   11847  /usr/sbin/apache2 -k start
apache   11848  /usr/sbin/apache2 -k start
apache   11849  /usr/sbin/apache2 -k start
apache   11850  /usr/sbin/apache2 -k start
apache   11851  /usr/sbin/apache2 -k start
```

If you go to the `log/` folder, you should see two new log files:

```
dev:/opt/modsecurity/var/log# l
total 1684
drwx------ 2 root root     4096 2009-08-20 10:31 .
drwxr-x--- 7 root apache   4096 2009-08-18 20:01 ..
-rw-r----- 1 root root   926530 2009-08-20 08:09 audit.log
-rw-r----- 1 root root   771903 2009-08-20 08:09 debug.log
-rw-r--r-- 1 root root      696 2009-08-20 10:33 mlogc-error.log
-rw-r--r-- 1 root root        0 2009-08-20 10:31 mlogc-transaction.log
```

If you look at the `mlogc-error.log` file, there will be signs of minimal activity (the timestamps from the beginning of every line were removed for clarity):

```
[3] [11893/0] Configuring ModSecurity Audit Log Collector 2.5.10-dev2.
[3] [11893/0] Delaying execution for 5000ms.
[3] [11895/0] Configuring ModSecurity Audit Log Collector 2.5.10-dev2.
[3] [11895/0] Delaying execution for 5000ms.
[3] [11893/0] Queue file not found. New one will be created.
[3] [11893/0] Caught SIGTERM, shutting down.
[3] [11893/0] ModSecurity Audit Log Collector 2.5.10-dev2 terminating normally.
[3] [11895/0] Queue file not found. New one will be created.
```

It is normal for two copies of mlogc to have run, because that's how Apache treats all piped logging programs. It starts two (one while it's checking configuration), but leaves only one running. The second token on every line in the example is the combination of process ID and tread ID. Thus you can see how there are two processes running at the same time (PID 11893 and PID 11895). Because only one program can handle the data files, `mlogc` is designed to wait for a while before it does anything. Basically, if it still lives after the delay, that means it's the copy that's meant to do something.

What happens if you make an error in the configuration file, which is preventing mlogc from working properly? As previously discussed, mlogc can't just respond to you on the

command line, so it will do the only thing it can. It will report the problem and shut down. (Don't be surprised if Apache continues with attempts to start it. That's what Apache does with piped logging programs.)

If you make a mistake in defining the error log, you may actually get an error message in response to the attempt to start Apache. Following is the error message you'd get if you left `ErrorLog` undefined:

```
dev:/opt/modsecurity/etc# apache2ctl start
[1] [12026/0] Failed to open the error log (null): Bad address
[3] [12026/0] ModSecurity Audit Log Collector 2.5.10-dev2 terminating with error 1
```

If `mlogc` managed to open its error log, it will do the expected and write all error messages there. For example:

```
[1] [11985/0] QueuePath not defined in the configuration file.
[3] [11985/0] ModSecurity Audit Log Collector 2.5.10-dev2 terminating with error 1
```

At this point, it is a good idea to delete the serial audit log file `audit.log`, or store it elsewhere. Having switched to concurrent logging, that file won't be updated any more and it will only confuse you.

Troubleshooting Mlogc

Assuming default logging configuration (level 3), a single audit log entry handled by mlogc will produce one line in the log file:

```
[3] [2435/693078] Entry completed (0.080 seconds, 9927 bytes):…
SsHOykMXI18AAAmnIgAAAABC
```

That's basically all you need to know—that an entry has been safely transmitted to the intended destination. Mlogc will treat the status codes 200 and 409 to indicate a successfully processed server. The response code 200 is used to indicate that there were no problems processing the entry; the response code 409 is used to indicate that there is a problem with the entry, but that it was processed nevertheless (and that mlogc should delete it and move on).

You will get more information when something gets wrong, of course. For example, you will see the following message whenever your logging server goes down.

```
[2] [2435/693078] Flagging server as errored after failure to submit entry…
SsHPNOMXI18AAAmLHucAAAAG (cURL code 7): couldn't connect to host
```

The message will appear on the first failed delivery, and then once every minute until the server becomes operational. This is because mlogc will shut down its operation for a short period whenever something unusual happens with the server. Only one thread of operation

will continue to work to probe the server, with processing returning to full speed once the server recovers. You'll get the following information in the log:

```
[3] [2435/693078] Clearing the server error flag after successful entry…
submission: SsHPNOMXI18AAAmLHucAAAAG
[3] [2435/693078] Entry completed (0.684 seconds, 9927 bytes):…
SsHPNOMXI18AAAmLHucAAAAG
```

Going back to the error message, the first part tells you that there's a problem with the server; the second part tells you what the problem is. In the previous case, the problem was "couldn't connect to host", which means the server is down.

Table 4.5. Common mlogc problems

Error message	Description
couldn't connect to host	The server could not be reached. It probably means the server itself is down, but it could also indicate a network issue. You can investigate the cURL return code to determine the exact cause of the problem.
Possible SSL negotiation error	Most commonly, this message will mean that you configured mlogc to submit over plain-text, but the remote server uses SSL. Make sure the `ConsoleURI` parameter starts with "https://".
Unauthorized	The credentials are incorrect. Check the `SensorUsername` and `SensorPassword` parameters.
For input string: "0, 0"	A remote server can indicate an internal error using any response status code other than 200 and 409, but such errors are treated as transient. ModSecurity Community Console has a long-standing problem where it responds with a 500 code to an audit log entry that is invalid in some way. The use of the 500 response code makes mlogc pause and attempt to deliver again, only to see the Console fail again. The process continues indefinitely and the only solution at the moment is to track down the offending audit log entry on the sensor and manually delete it.

If you still can't resolve the problem, I suggest that you increase the mlogc error log level from 3 (`NOTICE`) to 5 (`DEBUG2`), restart Apache (graceful will do), and try to uncover more information that would point to a solution. Actually, I advise you to perform this exercise even before you encounter a problem, because an analysis of the detailed log output will give you a better understanding of how mlogc works.

File Upload Interception

File upload interception is a special form of logging, in which the files being uploaded to your server are intercepted, inspected, and stored, and all that before they are seen by an

application. The directives related to this feature are in Table 4.6, "File upload directives", but you've already seen them all in the section called "File Uploads".

Table 4.6. File upload directives

Directive	Description
`SecUploadDir`	Specifies the location where intercepted files will be stored
`SecUploadFileMode`	Specifies the permissions that will be used for the stored files
`SecUploadKeepFiles`	Specifies whether to store the uploaded files (`On`, `Off`, or `RelevantOnly`)

Storing Files

Assuming the default configuration suggested in this guide, you only need to change the setting of the `SecUploadKeepFiles` directive to `On` to start collecting uploaded files. If, after a couple file upload requests, you examine `/opt/modsecurity/var/upload`, you'll find files with names similar to these:

```
20090818-164623-SorMz38AAAEAAFG2AOAAAAAA-file-okOc4T
20090818-164713-SorNAX8AAAEAAFG4AbUAAAAC-file-2ef1eC
```

You can probably tell that the first two parts of a filename are based on the time of upload, then follows the unique transaction ID, the `-file-` part that is always the same, and a random string of characters at the end. ModSecurity uses this algorithm to generate file names primarily to avoid filename collision and support the storage of a large number of files in a folder. In addition, avoiding the use of a user-supplied file name prevents a potential attacker from placing a file with a known name on a server.

When you store a file like this, it is just a file and it doesn't tell you anything about the attacker. Thus, for the files to be useful, you also need to preserve the corresponding audit log entries, which will contain the rest of the information.

> **Note**
>
> Storage of intercepted files can potentially consume a lot of disk space. If you're doing it, you should at least ensure the filesystem that you're using for storage is not the root filesystem—you don't want an overflow to kill your entire server.

Inspecting Files

For most people, a more reasonable `SecUploadKeepFiles` setting is `RelevantOnly`, which enables the storage of only the files that have failed inspection in some way. For this setting to

make sense, you need to have at least one external inspection script along with a rule that invokes it.

A file inspection rule is rather simple:

```
SecRule FILES_TMPNAMES "@inspectFile /opt/modsecurity/bin/file-inspect.pl" \
    phase:2,t:none,log,block
```

This example rule will invoke the script `/opt/modsecurity/bin/file-inspect.pl` for every uploaded file. The script will be given the location of the temporary file as its first and only parameter. It can do whatever it wants with the contents of the file, but it is expected to return a single line of output that consists of a verdict (`1` if everything is in order and `0` for a fault), followed by an error message. For example:

```
1 OK
```

Or:

```
0 Error
```

Following are the debug log lines produced by the inspection file:

```
[4] Recipe: Invoking rule 99e6538; [file "/opt/modsecurity/etc/rules.conf"]…
[line "3"].
[5] Rule 99e6538: SecRule "FILES_TMPNAMES" "@inspectFile…
/opt/modsecurity/bin/file-inspect.pl" "phase:2,auditlog,t:none,log,block"
[4] Transformation completed in 2 usec.
[4] Executing operator "inspectFile" with param…
"/opt/modsecurity/bin/file-inspect.pl" against FILES_TMPNAMES:f.
[9] Target value:…
"/opt/modsecurity/var/tmp//20090819-181304-SowyoH8AAQEAACW1AIo-file-ZPtFAq"
[4] Executing /opt/modsecurity/bin/file-inspect.pl to inspect…
/opt/modsecurity/var/tmp//20090819-181304-SowyoH8AAQEAACW1AIoAAAAA-file-ZPtFAq.
[9] Exec: /opt/modsecurity/bin/file-inspect.pl
[4] Exec: First line from script output: "1 OK"
[4] Operator completed in 6969 usec.
[4] Rule returned 0.
```

If an error occurs, for example if you make a mistake in the name of the script, you'll get an error message that looks similar to this one:

```
[9] Exec: /opt/modsecurity/bin/file_inspect.pl
[1] Exec: Execution failed while reading output:…
/opt/modsecurity/bin/file_inspect.pl (End of file found)
```

Integrating with ClamAV

ClamAV (`http://www.clamav.net`) is a popular open source anti-virus program. If you have it installed, the following script will allow you to utilize it to scan files from ModSecurity:

```
#!/usr/bin/perl

$CLAMSCAN = "/usr/bin/clamscan";

if (@ARGV != 1) {
    print "Usage: modsec-clamscan.pl FILENAME\n";
    exit;
}

my ($FILE) = @ARGV;

$cmd = "$CLAMSCAN --stdout --disable-summary $FILE";
$input = `$cmd`;
$input =~ m/^(.+)/;
$error_message = $1;

$output = "0 Unable to parse clamscan output";

if ($error_message =~ m/: Empty file\.$/) {
    $output = "1 empty file";
}
elsif ($error_message =~ m/: (.+) ERROR$/) {
    $output = "0 clamscan: $1";
}
elsif ($error_message =~ m/: (.+) FOUND$/) {
    $output = "0 clamscan: $1";
}
elsif ($error_message =~ m/: OK$/) {
    $output = "1 clamscan: OK";
}

print "$output\n";
```

> **Note**
>
> If you need an file to test with, you can download one from `http://www.eicar.org/anti_virus_test_file.htm`. The files at this location contain a test signature that will be picked up by ClamAV.

The error message from the integration script will either return the result of the inspection of the file or an error message if the inspection process failed. Following is an example that shows a successful detection of a "virus":

```
[9] Exec: /opt/modsecurity/bin/modsec-clamscan.pl
[4] Exec: First line from script output: "0 clamscan: Eicar-Test-Signature"
[4] Operator completed in 2137466 usec.
[2] Warning. File…
"/opt/modsecurity/var/tmp//20090819-181833-Sowz6X8AAQEAACXQAWAAAAAB-file-logg59"…
rejected by the approver script "/opt/modsecurity/bin/modsec-clamscan.pl": 0…
clamscan: Eicar-Test-Signature [file "/opt/modsecurity/etc/rules.conf"] [line "3"]
```

If you look carefully at the example output, you'll see that the inspection took in excess of 2 seconds. This is not unusual (even for my slow virtual server), because we're creating a new instance of the ClamAV engine for every inspection. The scanning alone is fast, but the initialization takes considerable time. A more efficient method would be to use the ClamAV daemon (e.g., the `clamav-daemon` package on Debian) for inspection. In this case, the daemon is running all the time, and the script is only informing it that it needs to inspect a file.

Assuming you've followed the recommendation for the file permissions settings given in the section called "Folder Locations", this is what you need to do:

1. Change the name of the ClamAV script from `clamscan` to `clamdscan` (note the added `d` in the filename).
2. Add the ClamAV user (typically `clamav`) to the group `apache`.
3. Relax the default file permissions used for uploaded files to allow group read, by changing `SecUploadFileMode` from `0600` to `0640`.

An examination of the logs after the change in the configuration will tell you that there's been a significant improvement—from seconds to milliseconds:

```
[9] Exec: /opt/modsecurity/bin/modsec-clamscan.pl
[4] Exec: First line from script output: "0 clamscan: Eicar-Test-Signature"
[4] Operator completed in 13329 usec.
[2] Warning. File…
"/opt/modsecurity/var/tmp//20090819-182404-Sow1NH8AAQEAACiEAIcAAAAA-file-AMzbgK"…
rejected by the approver script "/opt/modsecurity/bin/modsec-clamscan.pl": 0…
clamscan: Eicar-Test-Signature [file "/opt/modsecurity/etc/rules.conf"] [line "3"]
```

Guardian Log

The `SecGuardianLog` directive controls the functionality that establishes a bridge between the web server in which ModSecurity runs and an external process that wants to be notified about every transaction. The name comes from the name of the script that was the first (and

only, as far as I know) to use the directive: `httpd-guardian` is a part of my Apache httpd Tools project (see `http://www.apachesecurity.net/tools/` for more information).

Advanced Logging Configuration

By now you have seen how you have many facilities you can use to configure logging to work exactly as you need it. The facilities can be grouped into four categories:

Static logging configuration
: The various audit logging configuration directives establish the default (or static) audit logging configuration. You should use this type of configuration to establish what you want to happen in most cases. You should then use the remaining configuration mechanisms (listed next) to create exceptions to handle edge cases.

Setting of the relevant flag on rule matches
: Every rule match, unless suppressed, increments the transaction's *relevant* flag. This handy feature, designed to work with the `RelevantOnly` setting of `SecAuditEngine`, allows you to trigger transaction logging when something unusual happens.

Per-rule logging suggestions
: Rule matching and the actions `auditlog` and `noauditlog` do not control logging directly. They should be viewed as mere suggestions—it is up to the engine to decide whether to log a transaction. They are also ephemeral, affecting only the rules with which they are associated. They will be forgotten as the processing moves onto the next rule.

Dynamic logging configuration
: Rules can make logging decisions that affect entire decisions (through the `ctl` action), but that functionality should not be used lightly. Most rules should be concerned only with event generation, restricting their decisions to the suggestions mentioned in the previous. The ability to affect transaction logging should be used by system rules placed in phase 5 and written specifically for the purpose of logging control.

Increasing Logging from a Rule

Using the `SecAuditLogParts` directive, you will configure exactly what parts (how much information) you want logged for every transaction, but one setting will not be adequate in all cases. For example, most configurations will not be logging response bodies, but that information is often required to determine whether certain types of attack (XSS, for example) were successful.

The following rule will detect only simple XSS attacks, but when it does, it will cause the transaction's response body to be recorded:

```
SecRule ARGS <script> phase:2,log,block,ctl:auditLogParts=+E
```

Dynamically Altering Logging Configuration

The feature discussed in the previous section is very useful, but you may not always like the fact that some rules are changing what you're logging. I know I would not. Luckily, it's a problem that can be resolved with an addition of a phase 5 rule that resets the logged audit log parts:

```
SecAction phase:5,nolog,pass,ctl:auditLogParts=ABCDFGH
```

You can then decide on your own whether the logging of part `E` is justified. If you are using full audit logging in particular, you will need to manually increase the amount you log per transaction. The `HIGHEST_SEVERITY` variable, which contains the highest severity of a rule that matched during a transaction is particularly useful:

```
SecRule HIGHEST_SEVERITY "@le 2" phase:5,nolog,pass,ctl:auditLogParts=+E
```

Removing Sensitive Data from Audit Logs

Most web application programmers are thought to always use POST methods for the transactions that contain sensitive data. After all it is well known that request bodies are never logged, meaning that the sensitive data will never be logged, either. ModSecurity changes this situation, because it allows for full transaction logging. To deal with the sensitive data that may find its way into the logs, ModSecurity uses the sanitation actions `sanitiseArg`, `sanitiseRequestHeader` and `sanitiseResponseHeader`, and `sanitiseMatched`. You basically just need to tell ModSecurity which elements of a transaction you want removed and it will remove them for you, replacing their values in the log with asterisks. The first three actions all require parameters that you will typically know at configuration time, which means that you will invoke them unconditionally with `SecAction`. Sanitation works when invoked from any phase, but you should always use the phase 5, which is designed for this type of activity.

Use `sanitiseArg` to prevent the logging of the parameters whose names you know. For example, let's assume that you have an application that uses the parameters `password`, `oldPassword`, and `newPassword` to transmit, well, passwords. This is what you'll do:

```
SecAction phase:5,nolog,pass,\
    sanitiseArg:password,\
    sanitiseArg:oldPassword,\
```

```
    sanitiseArg:newPassword
```

Similarly, use sanitiseRequestHeader and sanitiseResponseHeader to remove the contents of the headers whose names you know. For example, if you have an application that uses HTTP Basic Authentication, you will need the following rule to prevent the passwords from being logged:

```
SecAction phase:5,nolog,pass,\
    sanitiseRequestHeader:Authorization
```

The last action, sanitiseMatched, is used when you need to sanitize a parameter whose name you don't know in advance. My first example will sanitize the contents of every parameter that has the word password in the name:

```
SecRule ARGS_NAMES password phase:5,nolog,pass,\
    sanitiseMatched
```

In the following example, we look for anything that resembles a credit card number and then sanitize it:

```
SecRule ARGS @verifyCC phase:5,nolog,pass,\
    sanitiseMatched
```

Selective Audit Logging

Although full transaction logging (the logging of every single byte of every single transaction) sounds good in theory, in practice it is very difficult to use, because it slows down your sever and requires huge amounts of storage space. There are ways to get some of the same benefits for a fraction of cost by using partial full logging on demand.

The trick is to tie in logging to IP address, user, or session tracking. By default you will log only what is relevant, but when you spot something suspicious coming from (for example) an IP address, you may change your logging configuration to turn on full logging for the offending IP address only. Here's how.

First you need to set up IP address tracking. You do this only once for all your rules, so it should usually be part of your main configuration:

```
SecAction phase:1,nolog,pass,initcol:ip=%{REMOTE_ADDR}
```

Now you need to add a phase rule that will trigger logging when something else happens. In the following case, we want to start logging everything coming from an IP address after a single rule match. To achieve that, we set the flag `ip.logflag` for up to one hour (3600 seconds):

```
SecRule HIGHEST_SEVERITY "@gt 0" \
```

```
    phase:5,nolog,pass,setvar:ip.logflag=1,expirevar:ip.logflag=3600
```

Finally, we add a rule that detects the flag and forces logging:

```
SecRule IP:logflag "@gt 0" \
    phase:5,nolog,pass,ctl:auditLogEngine=On
```

5 Rule Language Overview

The Rule Language is implemented using 9 directives, which are listed in Table 5.1, "Rule language directives".

Table 5.1. Rule language directives

Directive	Description
`SecAction`	Performs an unconditional action. This directive is essentially a rule that always matches.
`SecDefaultAction`	Specifies the default action list, which will be used in the rules that follow.
`SecMarker`	Creates a marker that can be used in conjunction with the `skipAfter` action. A marker creates a rule that does nothing, but has an ID assigned to it.
`SecRule`	Creates a rule.
`SecRuleInheritance`	Controls whether rules are inherited in a child configuration context.
`SecRuleRemoveById`	Removes the rule with the given ID.
`SecRuleRemoveByMsg`	Removes the rule whose message matches the given regular expression.
`SecRuleScript`	Creates a rule implemented using Lua.
`SecRuleUpdateActionById`	Replaces the action list of the rule with the given ID with the supplied action list.

The main directive to know is `SecRule`, which is used to create rules and thus does most of the work. The remainder of this section documents the individual elements that make the rules.

Anatomy of a Rule

Every rule defined by `SecRule` conforms to the same format, as below:

```
SecRule VARIABLES OPERATOR [TRANSFORMATION_FUNCTIONS, ACTIONS]
```

You can see all 4 building blocks of the rule language on the list. The 2 building blocks at the end are optional; if they are not explicitly defined in a rule, the defaults (inherited from a previous `SecDefaultAction` directive) will be used. So what do those building blocks do? Here's what:

Variables
: Identify parts of a HTTP transaction each rule works with. ModSecurity will extract information from every transaction and make it available, through variables, to rules to use. The important thing about variables to remember is that they are *binary strings*, meaning they can contain special characters and bytes of any value. Your sites may be restricting themselves to using only text in parameters, but that does not mean your adversaries will. In fact, your adversaries will use whatever helps them achieve their goals. A rule must specify one or more variables.

Operators
: Specify how is a (transformed) variable to be analyzed. Regular expressions are the most popular choice, but ModSecurity supports many other operators, and you are even able to write your own. Only one operator is allowed per rule.

Transformation functions
: A list of transformation functions that can be specified for every rule gives ModSecurity instructions how each variable is to be changed before analysis can be done. Transformation functions are commonly used to counter evasion, but they can also be used to uncover data that was encoded in some way.

Actions
: Specify what should be done when a rule matches.

Variables

In ModSecurity, variables are used to identify the exact place you want to look at in a HTTP transaction. One of the main features of ModSecurity is the fact that it pre-process raw transaction data and makes it easy for the rules to focus on the logic of detection. There are 77 variables in the most recent version of ModSecurity; they are listed in Table 5.2, "Request variables".

Regular variables
: Contain only one piece of information, or one string. For example, `REMOTE_ADDR`, always contains the IP address of the client.

Collections
: Groups of regular variables. Some collections (e.g., ARGS) allow enumeration, making it possible to use its every member in a rule. Some other collections (e.g., ENV)

are not as flexible, but there is always going to be some way to extract individual regular variables out of them.

Read-only collections
: Many of the collections point to some data that cannot be modified, in which case the collection itself will be available only for reading.

Read/write collections
: When a collection is not based on immutable data ModSecurity will allow you to modify it. A good example of a read/write collection is `TX`, which a collection that starts empty and exists only as long as the currently processed transaction exists.

Special collections
: Sometimes a collection is just a handy mechanism to retrieve information from something that is not organised as a collection but it can seem that way. This is the case with the XML collection, which takes an XPath expression as a (mandatory) parameter and allows you to extract values out of an XML file.

Persistent collections
: Some collections can be stored and retrieved later. This feature allows you to adopt a wider view of your systems, for example tracking access per IP address or per session, or per user account.

Request variables

Request variables are those extracted from the request part of the transaction that is being inspected. The variables that describe the request line (request method, URI and protocol information) and the request headers become available as early as phase 1 and the complete information will be available for phase 2.

Table 5.2. Request variables

Variable	Description
ARGS	Request parameters (read-only collection)
ARGS_COMBINED_SIZE	Total size of all request parameters combined
ARGS_NAMES	Request parameters' names (collection)
ARGS_GET	Query string parameters (read-only collection)
ARGS_GET_NAMES	Query string parameters' names (read-only collection)
ARGS_POST	Request body parameters (read-only collection)
ARGS_POST_NAMES	Request body parameters' names (read-only collection)
FILES	File names (read-only collection)
FILES_COMBINED_SIZE	Combined size of all uploaded files
FILES_NAMES	File parameter names (read-only collection)
FILES_SIZES	A list of file sizes (read-only collection)
FILES_TMPNAMES	A list of temporary file names (read-only collection)
PATH_INFO	Extra path information
QUERY_STRING	Request query string
REMOTE_USER	Remote user
REQUEST_BASENAME	Request URI basename
REQUEST_BODY	Request body
REQUEST_COOKIES	Request cookies (read-only collection)
REQUEST_COOKIES_NAMES	Request cookies' names (read-only collection)
REQUEST_FILENAME	Request URI filename/path
REQUEST_HEADERS	Request headers (collection, read-only)
REQUEST_HEADERS_NAMES	Request headers' names (read-only collection)
REQUEST_LINE	Request line
REQUEST_METHOD	Request method
REQUEST_PROTOCOL	Request protocol
REQUEST_URI	Request URI, convert to exclude hostname
REQUEST_URI_RAW	Request URI, as it was presented in the request

Server variables

Server variables contain the pieces of information available to the server, most of them valid only for the transaction being processed at the moment they are evaluated.

Table 5.3. Server variables

Variable	Description
AUTH_TYPE	Authentication type
REMOTE_ADDR	Remote address
REMOTE_HOST	Remote host
REMOTE_PORT	Remote port
SCRIPT_BASENAME	Script basename
SCRIPT_FILENAME	Script filename/path
SCRIPT_GID	Script group ID
SCRIPT_GROUPNAME	Script group name
SCRIPT_MODE	Script permissions
SCRIPT_UID	Script user ID
SCRIPT_USERNAME	Script user name
SERVER_ADDR	Server address
SERVER_NAME	Server name
SERVER_PORT	Server port

Response variables

Response variables are those extracted from the response part of the transaction that is being inspected. Most response variables will be available in phase 3. The arguably most important response variable, RESPONSE_BODY, is only available in phase 4 (the phase is also called RESPONSE_BODY).

Table 5.4. Response variables

Variable	Description
RESPONSE_BODY	Response body
RESPONSE_CONTENT_LENGTH	Response content length
RESPONSE_CONTENT_TYPE	Response content type
RESPONSE_HEADERS	Response headers (read-only collection)
RESPONSE_HEADERS_NAMES	Response headers' names (read-only collection)
RESPONSE_PROTOCOL	Response protocol
RESPONSE_STATUS	Response status code

Miscellaneous variables

Miscellaneous variables are exactly what they are called: they are the variables that couldn't fit in any other category.

Table 5.5. Utility variables

Variable	Description
HIGHEST_SEVERITY	Highest severity encountered
MATCHED_VAR	Contents of the last variable that matched
MATCHED_VAR_NAME	Name of the last variable that match
MODSEC_BUILD	ModSecurity build version (e.g., 02050102)
SESSIONID	Session ID associated with current transaction
USERID	User ID associated with current transaction
WEBAPPID	Web application ID associated with current transaction
WEBSERVER_ERROR_LOG	Error messages generated by Apache during current transaction

Parsing flags

Parsing flags are used by ModSecurity to signal important parsing events. The idea is to avoid taking implicit action (e.g., blocking in response to an invalid request), but allow the rules to decide what to do.

Table 5.6. Request body parsing variables

Variable	Description
MULTIPART_BOUNDARY_QUOTED	Multipart parsing error: quoted boundary encountered
MULTIPART_BOUNDARY_WHITESPACE	Multipart parsing error: whitespace in boundary
MULTIPART_CRLF_LF_LINES	Multipart parsing error: mixed line endings used
MULTIPART_DATA_BEFORE	Multipart parsing error: seen data before first boundary
MULTIPART_DATA_AFTER	Multipart parsing error: seen data after last boundary
MULTIPART_HEADER_FOLDING	Multipart parsing error: header folding used
MULTIPART_LF_LINE	Multipart parsing error: LF line ending detected
MULTIPART_SEMICOLON_MISSING	Multipart parsing error: missing semicolon before boundary
MULTIPART_STRICT_ERROR	At least one multipart error except MULTIPART_UNMATCHED_BOUNDARY occurred
MULTIPART_UNMATCHED_BOUNDARY	Multipart parsing error: unmatched boundary detected (prone to false positives)
REQBODY_PROCESSOR	Request processor that handled request body
REQBODY_PROCESSOR_ERROR	Request processor error flag (0 or 1)
REQBODY_PROCESSOR_ERROR_MSG	Request processor error message

Collections

Collections are the special kind of variables that can contain other variables. With exception of the persistent collections, all collections are essentially one-offs, special variables that give access to the information to which ModSecurity has access.

Table 5.7. Special collections

Variable	Description
ENV	Environment variables (read-only collection, although it's possible to use `setvar` to change it)
GEO	Geo lookup information from the last `@geoLookup` invocation (read-only collection)
GLOBAL	Global information, shared by all processes (read/write collection)
IP	IP address data storage (read/write collection)
TX	Transient transaction data (read/write collection)
RULE	Current rule metadata (read-only collection)
SESSION	Session data storage (read/write collection)
USER	User data storage (read/write collection)
XML	XML DOM tree (read-only collection)

Time variables

Time variables all represent the moment in time when the transaction that ModSecurity is processing began.

Table 5.8. Time variables

Variable	Description
TIME	Time (HH:MM:SS)
TIME_DAY	Day of the month (1-31)
TIME_EPOCH	Seconds since January 1, 1970 (e.g., 1251029017)
TIME_HOUR	Hour of the day (0-23)
TIME_MIN	Minute of the hour (0-59)
TIME_MON	Month of the year (0-11)
TIME_SEC	Second of the minute (0-59)
TIME_WDAY	Week day (0-6)
TIME_YEAR	Year

Operators

In the examples so far the assumption was that we are always going to use regular pattern matching against input. While regular expressions are very useful, there are often times when you want to do something else. That is when operators come to play. The truth is that ModSecurity always uses an operator, but that it assumes that you want to use regular patterns matching unless you specify an operator- -the regular pattern matching one!

```
SecRule ARGS:username "@rx ^(admin|root)$"
```

The above rule which checks if the requested username is `admin` or `root`. You may have noticed a few things:

- Operators begin with a @ character.
- Operators are always placed at the beginning of the second `SecRule` token.
- There's always a space after an operator. Whatever follows the space is the single operator parameter. In the case of the `@rx` operator, the parameter is a regular expression.
- When you have a rule with an explicit operator you'll need to use double quotes around the token, because there's always going to be a space character inside the token. Omitting double quotes would only confuse Apache and cause it to complain.

The power of the operators comes from the extensive functionality on offer. String operators, shown in Table 5.9, "String matching operators", are most often used. You've seen plenty examples using regular pattern matching so far. In addition to the simple matching operators (`@beginsWith`, `@endsWith`, etc.), ModSecurity supports parallel matching, where a large number of patterns can be matched at once. That is what `@pm` and `@pmFromFile` are for.

String matching operators

String matching operators all take a string on input and attempt to match it to the provided parameter. The `@rx` and `@pm` operators are the ones commonly used, because of their versatility (`@rx`) and speed (`@pm`), but the remaining operators are also useful, especially if you need variable expansion, which neither `@rx` nor `@pm` support.

Table 5.9. String matching operators

Operator	Description
`@beginsWith`	Begins with
`@contains`	Contains
`@endsWith`	Ends with
`@rx`	Regular pattern match
`@pm`	Parallel matching
`@pmFromFile` (`@pmf` in v2.6)	Parallel matching, with arguments from a file
`@streq`	String equal to
`@within`	Within

Numerical operators

Numerical operators, in Table 5.10, "Numerical operators" make comparing numerical values easy (previously you had to resort to using complex regular expressions).

Table 5.10. Numerical operators

Operator	Description
`@eq`	Equal
`@ge`	Greater or equal
`@gt`	Greater than
`@le`	Less or equal
`@lt`	Less than

Validation operators

Validation operators, in Table 5.11, "Validation operators", all validate input in some way.

Table 5.11. Validation operators

Operator	Description
`@validateByteRange`	Validates that parameter consists only of allowed byte values
`@validateDTD`	Validates XML payload against a DTD
`@validateSchema`	Validates XML payload against a Schema
`@validateUrlEncoding`	Validates an URL-encoded string
`@validateUtf8Encoding`	Validates an UTF-8 encoded string

Miscellaneous operators

And, finally, there's the miscellaneous category (Table 5.12, "Miscellaneous operators"), which offers some very useful functionality.

Table 5.12. Miscellaneous operators

Operator	Description
`@geoLookup`	Determines the physical location of an IP address
`@inspectFile`	Invokes an external script to inspect a file
`@rbl`	Looks parameter against a RBL (real-time block list)
`@verifyCC`	Checks if the parameter is a valid credit card number

Actions

Actions make ModSecurity tick. They make it possible to react to events and, more importantly, they are the glue that hold everything else together and make the advanced features possible. They are also the most overloaded element of the rule language. Because of the constraints of the Apache configuration syntax, within the rule language exists, actions are used to carry everything other than variables and operators. Actions can be split into 7 categories.

Disruptive actions

Disruptive actions (Table 5.13, "Disruptive actions") specify what a rule wants to do on a match. Each rule must be associated with exactly one disruptive action. The `pass` action is the only exception, as it will allow processing to continue when a match occurs. All other actions from this category will block in some specific way.

Table 5.13. Disruptive actions

Action	Description
`allow`	Stop processing of one or more remaining phases
`block`	Indicates that a rule wants to block
`deny`	Block transaction with an error page
`drop`	Close network connection
`pass`	Do not block, go to the next rule
`proxy`	Proxy request to a backend web server
`redirect`	Redirect request to some other web server

Flow actions

Flow actions (Table 5.14, "Flow actions") alter the way rules are processed within a phase.

Table 5.14. Flow actions

Action	Description
`chain`	Connect two or more rules into a single logical rule
`skip`	Skip over one or more rules that follow
`skipAfter`	Skip to the rule or marker with the provided ID

Metadata actions

Metadata actions (Table 5.15, "Metadata actions") provide additional information about rules. The information is meant to accompany the error messages to make it easier to understand why they occurred.

Table 5.15. Metadata actions

Action	Description
`id`	Assign unique ID to a rule
`phase`	Phase for a rule to run in
`msg`	Message string
`rev`	Revision number
`severity`	Severity
`tag`	Tag

Variable actions

Variable actions (Table 5.16, "Variable actions") deal with variables. They allow you to set, change and remove variables.

Table 5.16. Variable actions

Action	Description
capture	Capture results into one or more variables
deprecatevar	Decrease numerical variable value over time
expirevar	Remove variable after a time period
initcol	Create a new persistent collection
setenv	Set or remove an environment variable
setvar	Set, remove, increment or decrement a variable
setuid	Associate current transaction with an application user ID (username)
setsid	Associate current transaction with an application session ID

Logging actions

Logging actions (Table 5.17, "Logging actions") influence the way logging is done. The actions that influence if logging takes place (auditlog, log, noauditlog, and nolog) only control current rule affects logging if it matches. To control logging for the transaction as a whole you'll need to use the ctl action.

Table 5.17. Logging actions

Action	Description
auditlog	Log current transaction to audit log
log	Log error message; implies auditlog
logdata	Log supplied data as part of error message
noauditlog	Do not log current transaction to audit log
nolog	Do not log error message; implies noauditlog
sanitiseArg	Remove request parameter from audit log
sanitiseMatched	Remove parameter in which a match occurred from audit log
sanitiseRequestHeader	Remove request header from audit log
sanitiseResponseHeader	Remove response header from audit log

Special actions

Special actions (Table 5.18, "Special actions") are gateways of sort; they provide access to another class of functionality. The ctl action has several sub-actions of its own and allows engine configuration to be changed only the current transaction. The multiMatch rule activates a special way of matching in which the rule operator is run after every transformation

(normally, the operator is run only once after all transformations). The `t` action is used to specify zero or more transformations that will be applied to variables before an operator is run.

Table 5.18. Special actions

Action	Description
`ctl`	Change configuration of current transaction
`multiMatch`	Activate multi-matching, where an operator runs after every transformation
`t`	Specify transformation functions to apply to variables before matching

Miscellaneous actions

`Miscellaneous actions` (Table 5.19, "Miscellaneous actions") contain the actions that don't belong in any of the groups.

Table 5.19. Miscellaneous actions

Action	Description
`append`	Append content to response body
`exec`	Execute external script
`pause`	Pause transaction
`prepend`	Prepend content to response body
`status`	Specify response status code to use with `deny` and `redirect`
`xmlns`	Specify name space for use with XPath expressions

6 Rule Language Tutorial

Now that you have a basic understanding of what the rules look like, I will walk you through the most commonly used functionality using examples.

Introducing simple rules and operators

The simplest possible rule will specify only a variable and a regular expression. In the example that follows, we look at the request URI, trying to match the regular expression pattern `<script>` against it:

```
SecRule REQUEST_URI <script>
```

This simple rule takes advantage of the fact that ModSecurity allows a rule not to specify an operator, in which case it assumes the regular expression operator. This feature is a left-over from ModSecurity 1.x, which only supported regular expressions—there were no operators at all. If you wish to you can always explicitly specify the operator. I always do. The above rule is functionally identical to this one:

```
SecRule REQUEST_URI "@rx <script>"
```

Note how I've had to use double quotes because the second parameter now contains a space.

ModSecurity supports a number of operators. Some are similar, but often have different performance characteristics. For example, the regular expression pattern I used for the examples (`<script>`) above isn't much of a pattern. It's just a string, because it does not contain any special characters. I might have just as well written the same rule using the `@contains` operator:

```
SecRule REQUEST_URI "@contains <script>"
```

By now you are probably aware that the operators are very straightforward. They take a piece of a transaction and analyse it, typically comparing it in some way to the parameter you provided in the rule (`<script>` in the above examples).

Working with variables

You can specify as many variables in a rule as you wish for as long as you separate them using the pipe character:

```
SecRule REQUEST_URI|REQUEST_PROTOCOL <script>
```

Some variables, which we call *collections*, potentially contain more than one piece of information. This is the case with the ARGS variable, for example, which contains all request parameters in a transaction. You use the colon operator to specify only one member of a collection, as you can see in the following rule which only looks at the parameter named p:

```
SecRule ARGS:p <script>
```

You can use the same collection more than once within the same rule, if you wish:

```
SecRule ARGS:p|ARGS:q <script>
```

The colon operator is actually quite potent and allows you to use a regular expression to specify the names, which is helpful when parameter names change at run-time. The following rule will target all parameters whose names begin with the letter p, catching parameters such as password, or pea:

```
SecRule ARGS:/^p/ <script>
```

When you do not restrict a rule to only some members of a collection ModSecurity will assume you want to use all of them. This is quite handy to use in the cases where you don't know what parameters a page uses. Not all collections can be used in this way (for example, ARGS can but ENV cannot), but when they can a reference to such collection will be expanded into individual variables just before a rule is run. You can observe in the debug log how this works. For example, for a request that has the parameters p, q and z, ARGS expands as follows:

```
[4] Expanded "ARGS" to "ARGS:p|ARGS:q|ARGS:z".
```

Now that you know how expansion work, parameter exclusion will make sense: to remove a parameter from a rule just put an exclamation mark before it. The following rule will look at all request parameters except at the one named z:

```
SecRule ARGS|!ARGS:z <script>
```

Combining rules into chains

When you specify more than one variable in a rule you are effectively combining them using the OR logical operator. The rule will match if any of the variables matches. It is also possible

to use a logical AND, whereby you combine several rules into one. Let's say that you want to write a rule that matches when something is found in both the parameter `p` and the parameter `q`. You write:

```
SecRule ARGS:p <script> chain
SecRule ARGS:q <script>
```

This is called *rule chaining*. The `chain` action constructs a chain of two or more rules and effectively creates a single rule with more than one evaluation step. The first rule in a chain will always run, but the subsequent rules will run only if all the previous rules (in the same chain) ran. Whenever a rule that belongs to a chain does not match, the execution continues with the first rule that is not part of that chain.

Operator negation

Operator result can be negated by placing an exclamation mark right before it. For example, if you wanted to write a rule that matches on a username that is not `admin` nor `root` (the opposite of the intent in the previous example), you write this:

```
SecRule ARGS:username "!@rx ^(admin|root)$"
```

Operator negation should not be confused with rule negation. The two are the same only when a rule is used against only one variable, but the situation changes when there are more. Observe the following rule:

```
SecRule ARGS:p|ARGS:q "!@eq 5"
```

The above rule will match if any one parameter does not equal 5. If you want to write a rule that matches when both parameters do not equal 5 you'll have to use rule chaining:

```
SecRule ARGS:p "!@eq 5" chain
SecRule ARGS:q "!@eq 5"
```

Variable counting

Here's a question for you: how do you detect something that isn't there? Take the common rule that addresses all parameters in a request:

```
SecRule ARGS <script>
```

In a request without any parameters `ARGS` will expand to zero variables. Without any variables to work with any operator will fail and the rule (or a chain) will not match.

The answer is to use the ability of ModSecurity to count how many variables there are in a collection. With the help of the ampersand operator we can look into ARGS and detect the case when there are no parameters:

```
SecRule &ARGS "@eq 0"
```

The ampersand operator can be applied to any collection, including a partial one. The following rule will match whenever it sees a request with more than one parameter named username:

```
SecRule &ARGS:username "!@eq 1"
```

Using actions

Most of the examples in this tutorial, so far, didn't use any actions. I chose to initially focus only on the mechanics of detection. But it is practically impossible to write a rule without specifying a single action. Furthermore, it is good practice to write rules that are self-contained and do not rely on the defaults.

Actions are placed in the third parameter of SecRule and the first parameter of SecAction. A rule can have zero, one or more actions. If there is more than one action, they are separated with a comma and any number of whitespace characters in between. The following rule specifies two actions:

```
SecRule ARGS K1 log,deny
```

Some actions have parameters, in which case you will need to place a colon after the action name and follow with the parameter. To deny with status 404, you could use:

```
SecRule ARGS K1 log,deny,status:404
```

Finally, if you want to supply a parameter that uses whitespace or contains a comma, enclose the value in single quotes. This way of parameter handling is often needed with messages:

```
SecRule ARGS K1 "log,deny,msg:'Acme attack detected'"
```

In addition to using single quotes around the parameter to the msg action, I enclosed the entire third directive parameter in double quotes. This is needed for Apache to correctly parse the directive line. You shall see later that some actions take complex parameters (e.g., ctl and setvar), but the same syntax discussed here applies to them too.

Understanding action defaults

You now know how to specify rule actions, but what happens if you don't? ModSecurity has a concept of default action list. Whenever a new rule is added to the configuration, the action list of the rule is merged with the default action list. The default action list is (in ModSecurity 2.5.11) is `phase:2,log,auditlog,pass`, but you can override that at any time using the `SecDefaultAction` directive. In the simplest case, when the rule being added has no action, the default action list is used instead. Take the following rule (and assume there are no other rules or defaults in the configuration):

```
SecRule ARGS K1
```

It is equivalent to:

```
SecRule ARGS K1 phase:2,log,auditlog,pass
```

In a general case, when a rule has one or more actions, merging means one of two things:

Rule action replaces an action in the default action list
: This will typically happen with disruptive actions, of which there can only be one per rule. If there's a disruptive action specified in the default actions list and in the rule, the one in the rule will override the default one

Rule action is appended to the ones in the default action list
: Some actions can appear more than once in an action list. This is the case with many non-disruptive actions, for example `t`, `setvar`, `ctl` and so on. In some cases it is possible for the rule actions to completely remove the default actions, but how that's done depends on the action in question. With transformation action, for example, specifying none as an transformation function (`t:none`) will clear the list and start over.

The idea with `SecDefaultAction` was to make the job of rule writing easier by enabling you to specify the commonly used actions only once. For example, you could write something like this:

```
SecDefaultAction phase:2,log,deny,status:404
SecRule ARGS K1
SecRule ARGS K2
...
SecRule ARGS K99
```

The above approach works well when you're in complete control of your configuration, but it complicates things because the rules are no longer self contained. The configuration is perhaps easier to write one day, but more difficult to understand when you come in a couple of months' time. Furthermore, there's always a danger that there will be unforeseen interaction between the defaults and the rule. For example, suppose you write a rule that rely

on certain default values, but then you later change the defaults without realising how you're impacting the rules. This is particularly true if you place any transformation functions in the default list:

```
SecDefaultAction phase:2,log,pass,t:lowercase
SecRule ARGS K1 t:urlDecode
```

> **Note**
>
> You should always write rules to specify the complete list of transformation functions they depend on. To achieve this, always specify `t:none` as the first transformation function, which will reset the transformation pipeline.

Another peculiarity with the `SecDefaultAction` directive, and that is that it can be used more than once. Every time you use it the default action list is changed. For example:

```
# First we have some rules that only warn
SecDefaultAction phase:2,log,pass
SecRule ARGS W1
SecRule ARGS W2
...
SecRule ARGS W19

# Now we have some rules that block
SecDefaultAction phase:2,log,deny,status:500
SecRule ARGS B1
...
SecRule ARGS B89
```

The bottom line is that, even though `SecDefaultAction` is quite powerful and allows you to specify any action, you should only use it to specify the default blocking method. Anything other than that is asking for trouble! Because of that, and because of some other issues that occur whenever `SecDefaultAction` is used in configuration with multiple contexts (which will be explained in the section called "SecDefaultAction inheritance anomaly"), there is a good probability that `SecDefaultAction` will be deprecated and replaced with a safer mechanism in the future.

Actions in chained rules

Special rules apply to the placement of actions in chained rules. Because several chained rules form a single complex rule, there can only be one disruptive action for the entire chain. Similarly, there can be only one set of meta data rules. By convention, the disruptive action and the meta data actions are placed with the first rule in a chain.

```
SecRule ARGS K1 chain,id:1001,log,deny
SecRule ARGS k2
```

The above example looks innocent enough, but trouble begins once you start to write complex chained rules (as most are), when you will have to mix non-disruptive actions with the disruptive ones. For example:

```
SecRule ARGS K1 chain,id:1001,log,deny,setvar:tx.score=+1
SecRule ARGS K2 setvar:tx.score=+1
```

When reading a complex chained rule you will have to remember that non-disruptive actions associated with a rule will always execute, even when a rule is part of a chain, but the disruptive actions will execute only when the last rule in the chain matches.

In hindsight, the last rule in a chain is a much better location for the disruptive and meta data rules, but that's too late to change now.

Unconditional rules

The actions you specify in a `SecRule` execute when a match occurs, but you can use the `SecAction` directive to do something unconditionally. This directive accepts only one parameter, identical the third parameter of `SecRule`, and it's a list of actions you want to be executed:

```
SecAction nolog,pass,setvar:tx.counter=10
```

The `SecAction` directive is useful in the following cases:

- To initialize one or more variables before the rules that use them are processed
- To initialize a persistent collection, most often using a client's IP address
- In combination with `skip`, to implement an if-then-else construct

Using transformation functions

You already know that rules typically work by taking some data, determined by a variable name, and applying an operator to it. But direct matching like that only happens in the simplest case. In a general case, the data processed by a rule will be transformed by one or more transformation functions before it is fed to an operator. The transformation functions are often referred to as a transformation pipeline.

As an example, take the following rule, which transforms input by converting all characters into lowercase, then compressing multiple consecutive whitespace characters:

```
SecRule ARGS "@contains delete from" \
```

```
    phase:2,t:lowercase,t:compressWhitespace,block
```

As a result, the rule will match all the following forms of input:

```
delete from
DELETE FROM
deLeTe fRoM
Delete   From
DELETE\tFROM -- \t represents a TAB character
```

> **Note**
>
> It is a good practice to always begin the list of transformation functions with `t:none`, which clears the transformation pipeline to start from scratch. If you don't do that then you, as a rule writer, can never be completely sure that your user didn't specify a transformation function in his `SecDefaultAction` directive (on purpose or by mistake), in which case your rule will probably malfunction. Using `t:none` ensures your rule always uses the correct set of transformation functions.

There are several reasons why you might want to do apply operators to something else than the original variable values:

- Your input is not available in a form that is useful to you. For example, it might be base64-encoded, in which case you won't be able to do anything useful with it. By applying the transformation function that decodes base64 data (`t:base64Decode`), you "open" up the data for inspection.
- Similarly, you may need a piece of data in some other form. If you have some binary data, which you need to record in a user-friendly manner, you will probably encode it as hex characters using `t:hexEncode`.
- Sometimes rules are difficult or impossible to write to deal with input in its original form. Take, for example, case sensitivity. Most ModSecurity operators are case sensitive, but there are many cases where case does not matter. If you attempt to match a non-trivial string using a case sensitive matching function, you will soon discover that you will either need to write a number or rules (each with a different combination of lowercase and uppercase letters) or a rule with a very ugly and difficult to decipher regular expression. You deal with this particular problem by transforming input into lowercase before matching.
- In the majority of cases, however, you will use transformation functions to counter evasion. Evasion is a technique often used by attackers to bypass existing detection and protection mechanism. They will take advantage of the specific context in which attack payload data is processed to modify it in such a way to evade detection, but remain effective.

Blocking

No matter if you use actions or not, every ModSecurity rule is always associated with one (and only one) disruptive action. The disruptive actions are those that interrupt rule processing within a phase. A disruptive action can do one of three things:

Continue with the next rule
: This is a special case of a disruptive action that doesn't disrupt. Use the `pass` action whenever you want to only warn about a potential issue, or if you want to have a rule that changes something else in the transaction or persistent state (e.g., increments a counter).

Stop processing phase but continue with transaction
: The `allow` action is used for whitelisting. It allows transactions to proceed without further inspection. Depending on how you use allow, you may choose to skip just the current phase, the request inspection phases (phases 1 and 2) and all remaining inspection phases (the logging phase always runs). Whitelisting is very important and I dedicate the entire the section called "Whitelisting" (Chapter 7) to it.

Stop processing phase and block transaction
: Blocking is a last-resort measure you undertake to either protect your web applications or turn away undesirable clients (e.g., worms, bots and similar). The best way for a rule to block is by using the `block` action, which indicates blocking but does not state how it is to be done. Another advantage of `block` is that it can be overridden by the rule administrator. If you use any of the other blocking actions (`deny`, `drop`, `redirect` and `proxy`) you are essentially hard-coding policy in rules. That may be all right if you are writing one-off rules for yourself, but be warned that, for others to use your rules, they will probably have to change them to suit their circumstances.

If you are very interested in blocking, head to the section called "Advanced Blocking", in Chapter 6, which covers the topic in detail.

Changing rule flow

The assumption with ModSecurity rules is that they will be processed one by one, starting with the first rule in a phase and ending with the last. If a match occurs somewhere in the phase and blocking takes place, phase processing will stop, but the execution of the rules is still linear. But there is only so much you can achieve executing rules in that fashion. Sometimes you will want to form rule groups and create if-then-else constructs, and for that you will need the actions that change the way rules flow.

Historically, the first skipping action supported by ModSecurity was `skip`, which takes one parameter and skips over as many rules as you specify. Rule skipping does not make any sense when used with a disruptive action, which means that you will only use `skip` in combination with `pass`. The following example demonstrates `skip`:

```
SecRule ARGS K1 id:1,nolog,pass,skip:2
SecRule ARGS K2 id:2,nolog,pass
SecRule ARGS K3 id:3,log,block
```

In the example above, when rule 1 matches it will skip the next two rules. It is as simple as that.

You should have the following in mind:

- When you form a chain of two or more individual rules, the entire chain counts as one rule for the sake of skipping.
- You can use `skip` in a chain, but the same rules as for the disruptive actions apply: only one skip is allowed and it has to be placed within the chain starter rule.
- Skipping takes place with rules that belong to the same phase as the rule that is initiating the skipping. You must not count any rules that belong to phases other than the one to which the skipping rule belongs. To avoid tricky situations, you shouldn't mix rule phases and the `skip` action.

Skipping is often used as an optimization technique. Sometimes executing a group of rules makes sense only under a specific condition and executing them otherwise is a waste of CPU power. In such cases you will typically precede the group with a single rule that tests for the condition and jumps over the entire group of rules if the condition is not true.

Over time, several problems were identified with the `skip` action. First, counting the rules you wish to skip over is not very interesting, and it's easy to make a mistake. It also makes maintenance difficult. Every time you want to make changes to your rules you have to first remember that you have a skipping rule in the neighbourhood, then look at it and figure if you need to update the skip parameter. There is also a potentially big problem that occurs when you use `SecRuleRemoveById` or `SecRuleRemoveByMsg` to remove a rule that is skipped over. With one rule less to skip over, the skip action will consume the intended next rule. The following example demonstrates this problem:

```
SecRule ARGS K1 id:1,nolog,pass,skip:1
SecRule ARGS K2 id:2,nolog,pass
SecRule ARGS K3 id:3,nolog,pass

...

SecRuleRemoveById 2
```

Rule 1 wants to skip over rule 2 on a match, but because we remove rule 2 later in the configuration, rule 1 will skip over rule 3 instead.

Smarter skipping

After identifying the problems with skip, we decided to improve the rule language slightly and added `skipAfter` and `SecLabel` to the rule language. The first example, rewritten to use the new facilities, looks like this:

```
SecRule ARGS K1 id:1,nolog,pass,skipAfter:4
SecRule ARGS K2 id:2,nolog,pass,skipAfter:4
SecRule ARGS K3 id:3,log,block
SecLabel id:4
```

When you use skipAfter, it will start to examine all the rules to follow to find the one with the specified ID. Once found, rule execution will continue with the next rule. This really means that you don't always need to use SecLabel. In many cases, skipAfter alone will work just fine. The same example can be rewritten like this:

```
SecRule ARGS K1 id:1,nolog,pass,skipAfter:3
SecRule ARGS K2 id:2,nolog,pass,skipAfter:3
SecRule ARGS K3 id:3,log,block
```

If-then-else

You can implement a primitive if-then-else construct if you use `skip` and `SecAction` together:

```
SecRule ARGS K1 id:1,nolog,pass,skip:2
SecRule ARGS K2 id:2,block
SecAction nolog,pass,skip:1
SecRule ARGS K3 id:3,block
```

The first rule in the example determines which of the two paths will be processed. If it matches, the skip action is executed to skip to rule 3. However, if it doesn't match the next rule (2) will be processed. The unconditional match in `SecAction`, which follows rule 1, ensures that rule 2 is not processed if there is no match in rule 1.

Skipping using labels doesn't make the rules easier to read, although it makes large rule groups easier to maintain:

```
SecRule ARGS K1 id:1,nolog,pass,skipAfter:11
SecRule ARGS K2 id:2,block
SecAction nolog,pass,skipAfter:12
```

```
SecLabel 11
SecRule ARGS K3 id:3,block
SecLabel 12
```

Controlling logging

There are several logging actions a rule can use, and they fall into two groups. (As a reminder, you can find the list of all logging rules in Table 5.17, "Logging actions".) The first group consist of the actions that influence only what happens during the processing of the current rule; such actions are used in virtually every rule and I cover them in this section. The actions in the second group influence how logging is done on a transaction level, and they are normally only used in configuration rules. I will not cover the second group here because the common use cases are already covered in the section called "Advanced Logging Configuration".

Going back to the first group, the most common usage is as follows:

```
SecRule ARGS K1 log,auditlog,block
```

If the above rule matches, the actions `log` and `auditlog` tell the engine the emit an alert and log the transaction to the audit log, respectively. I will let you in on a secret. The `log` action actually implies `auditlog`, so it is always safe to use only the first. (The same is true for the actions that ask for no logging: `nolog`, the opposite of `log`, implies noauditlog, which is the opposite of `auditlog`.) There are two things to consider:

1. An alert is a record of a rule match that will appear in the debug log, in the Apache's error log, and in the H section of an audit log entry. Because there are two pairs of actions (`log` and `nolog`, and `auditlog` and `noauditlog`) you can decide exactly what happens, logging-wise, when a rule matches. Most rules will want both, but you may equally log a match only to the error log and not have an entire audit log entry (which you achieve with `log` and `noauditlog` together).
2. When a rule specifies `auditlog` that does not mean that an audit log will be created. You should think about auditlog as *asking* for a transaction to be recorded, but a detection rule will not normally have full control over what will actually happen. ModSecurity classifies transactions as relevant or not relevant. When a rule matches and when it specifies `auditlog` (either explicitly, or implicitly through `log` without `noauditlog`), ModSecurity will set the relevancy flag. This will normally cause the transaction to be recorded, but, as we have seen in the section called "Advanced Logging Configuration", a subsequent rule can override that decision. This separation of concerns is intentional. Rules should only indicate what they want to achieve, but it is the administrator who should have the final say.

Capturing data

The TX collection has 10 variables whose names are just digits from 0 to 9. Those variables are reserved for *data capture*, which is primarily a feature of the @rx operator. To make use of this feature you have to do two things:

1. Use parenthesis within regular expression patterns to specify where capture should take place
2. Add the capture action to the rule in which you wish data capture to take place

Suppose you are dealing with a web application that places session identifiers in the request line. In order to support session state you will need to extract the session information and initialize session state. The URI used in the application and containing a session identifier could look like this:

```
http://www.example.com/69d032331009e7b0/index.html
```

Your rule to extract the session identifier will use a regular expression data capture:

```
# Initialize session state from the session identifier in URI
SecRule REQUEST_URI ^/([0-9a-fA-f]{16})/ phase:1,nolog,pass,capture,setsid:%{TX.1}
```

> **Note**
>
> Although the above example neatly demonstrates the data capture mechanism, that one rule alone is not enough for a correct implementation of session management. Refer to the section called "Session Management" for complete coverage.

Here is what happens on a successful match:

```
[4] Recipe: Invoking rule 8e8b5c8; [file "/home/ivanr/apache/conf/httpd.conf"] …
[line "479"].
[5] Rule 8e8b5c8: SecRule "REQUEST_URI" "@rx ^/([0-9a-fA-f]{16})/"
"phase:1,auditlog,nolog,pass,capture,setsid:%{TX.1}"
[4] Transformation completed in 2 usec.
[4] Executing operator "rx" with param "^/([0-9a-fA-f]{16})/" against REQUEST_URI.
[9] Target value: "/69d032331009e7b0/index.html"
[9] Added regex subexpression to TX.0: /69d032331009e7b0/
[9] Added regex subexpression to TX.1: 69d032331009e7b0
[4] Operator completed in 63 usec.
```

The TX.0 variable will always contain the entire part of the input that was matched (69d032331009e7b0 in the example; note the forward slashes at the beginning and at the end of the value). If your regular expression uses the ^ and $ anchors, TX.0 will contain the entire input. In the above example I only used one of the anchors, so TX.0 contains the data from the beginning of input but only until the end of the matching part (the second forward

slash). The TX.1 variable will contain just the part that was enclosed in the first parentheses set that appeared in the pattern. The TX.2 variable will draw its contents from the second set of parenthesis, and so on. Up to 9 captures will be created.

> **Note**
>
> If there is no match, the data capture variables will not be changed. However, if there is a match, the unused data capture variables will be unset.

The @pm and @pmFromFile operators have limited support for data capture: if the capture action was specified the TX.0 variable will be populated with the input data matched. There is no need to use parentheses in the patterns anywhere.

Variable manipulation

Although most of the data you're be dealing with will be read-only, generated by Apache and ModSecurity as they parse transaction data, there are certain variables and collections you are allowed to change. The TX collection is a private, per-transaction, space that rules can use to collaborate. The variables placed in TX can be retrieved using the same approach as for other collections. The setvar action, however, allows the values to be changed.

To create a new variable, simply set its value to something:

```
SecAction nolog,pass,setvar:tx.score=1
```

To delete a variable, place an exclamation mark before the name:

```
SecAction nolog,pass,setvar:!tx.score
```

Numerical values can be incremented or decremented. The following example first increments a variable by 2 then decrements it by 1:

```
SecAction nolog,pass,setvar:tx.score=+2
SecAction nolog,pass,setvar:tx.score=-1
```

Although collaboration within the same transaction is very interesting and useful, variable manipulation becomes exciting when combined with the persistent storage functionality and the expirevar and deprecatevar actions (covered in Chapter 8, *Persistent Storage*).

Variable expansion

In many text contexts, ModSecurity supports a feature known as variable expansion. The reference manual refers to it as macro expansion, but I think that is rather ambitious, at

least at this time. Variable expansion enables you to put data into text, which can be very useful. You may recall that I used variable expansion in the system rules in the section called "Handling Parsing Errors":

```
SecRule REQBODY_PROCESSOR_ERROR "!@eq 0" \
    "phase:2,t:none,log,block,msg:'Failed to parse request body: …
%{REQBODY_PROCESSOR_ERROR_MSG}',severity:2"
```

The idea above is that, when fault during request body parsing occurs, you are able to see what the actual error actually. Variable expansion takes place whenever ModSecurity encounters a variable name enclosed in `%{...}`, which is a syntax ModSecurity adopted from mod_rewrite. The variable name can be anything, and you are able to access collections using the familiar syntax `%{COLNAME.VARNAME}`.

> **Note**
>
> The difference between `COLNAME:VARNAME` and `COLNAME.VARNAME` is that the former potentially returns more than one result, whereas the latter will always return one result (or no result at all).

Most parts of the rule language support variable expansion; many features actually require it. For example, session or IP address tracking would be impossible without an ability to somehow handle a piece of data received from a client. Having said that, don't be surprised if you encounter a part of the rule language that does not support this feature. Initially, the support for this feature was added only to the places where it was needed. By popular demand the support expanded over time, but quite possibly not throughout. If that happens, you are advised to report the problem to the issue tracker.

Here's an interesting example that uses variable expansion, where one piece of a request is compared to another one, from the same request:

```
# If an absolute URI (containing hostname) was given on the request
# line, check that the same hostname is used in the Host header
SecRule REQUEST_URI_RAW "@beginsWith http" "chain,phase:2,block,msg:'Hostname …
mismatch'"
SecRule REQUEST_URI_RAW "!@beginsWith http://%{REQUEST_HEADERS.Host}"
```

Variable expansion is not supported by the regular expression and parallel matching operators, for performance reasons. Both `@rx` and `@pm` split their work into two steps. They do as much work as possible upfront, compiling patterns into more efficient internal representations. Then, in the second step they perform matching. The compilation of patterns is done only once, at configuration time, thus requiring the patterns to be static. On the upside, the matching is match faster. Variable expansion is where the string operators other than `@rx` and `@pm` have a rare advantage.

Recording data in alerts

The one remaining unmentioned log action is `logdata`, whose purpose is take a piece of data you specify and include it along with other alert information.

Consider the following rule, which looks for JavaScript event handlers in input:

```
SecRule ARGS "\bon(abort|blur|change|click|dblclick|dragdrop|end|error|\
focus|keydown|keypress|keyup|load|mousedown|mousemove|mouseout\
mouseover|mouseup|move|readystatechange|reset|resize|select|submit|unload)\b\W*?=" \
    phase:1,t:none,t:lowercase,log,deny,capture,logdata:%{TX.0}
```

This rule may seem a bit intimidating at a first glance, although it is conceptually simple. If you read the regular expression pattern carefully, you will see that all the patterns we are looking for share the beginning, have a part in the middle that is different, and share the end. So it's not that difficult after all. However, consider the following:

- Alert messages do not display input data. Thus, looking at an alert message alone, you will not be able to tell which part of the pattern match, and you will have to seek access to the entire audit log. Even when possible to get it, it will still be time consuming.
- Even with access to the audit log, tracking down the part of input that matched may not be simple. When this sort of rule matches, it typically happens with request parameters that are quite long. So what you'd need to do first understand what the rule does and then effectively perform manual pattern matching by reading through every parameter.
- Matching takes place against potentially transformed input, so often the raw input will not contain the data in the form used for matching.

These problems are resolved when you use the `logdata` action. Have a look at the following alert (just the emphasized part):

```
[Fri Dec 04 17:00:01 2009] [error] [client 192.168.3.1] ModSecurity: Access denied
with code 403 (phase 1). Pattern match "\\bon(abort|blur|change|click|dblclick
|dragdrop|end|error|focus|keydown|keypress|keyup|load|mousedown|mousemove
|mouseoutmouseover|mouseup|move|readystatechange|reset|resize|select
|submit|unload)\\b\\W*?=" at ARGS:p. [file "/home/ivanr/apache/conf/httpd.conf"]
[line "472"] [data "onload="] [hostname "192.168.3.100"] [uri "/"]
[unique_id "SxlAEcCoA2QAABLXHEAAAAAB"]
```

The `capture` action from the rule told the regular expression operator (`@rx`) to place the entire matching area into the variable `TX.0`. The `logdata:%{TX.0}` part of the rule told the engine to include the value of the `TX.0` variable in the alert. The end result is that you now know, at a glance, exactly what matched.

> **Note**
>
> At this point you may ask, why do we have `logdata`, when it is perfectly possible to use variable expansion in the `msg` action? There's only one reason: when you place a piece of data as part of the message, a programmatic parser will not know about that. To a computer, the entire message is just some text. But if you include the same data in an alert with `logdata`, the same parser will know that it is something that originated in input, and it can do something useful with it. It could, for example, highlight the piece of data on the alert page.

Adding meta data

While some rules are simple and do not require much thought to understand them, many aren't. Also, even when the rule itself is simple, that does not meant that it will be easy to understand what it does and why it does it. ModSecurity will generally try to add as much meta data to alerts as possible. Consider the following rule, which gets the job done:

```
SecRule REQUEST_METHOD "!^(GET|HEAD)$" \
    phase:1,t:none,log,block
```

It restricts request methods to either `GET` or `HEAD`, which is suitable only for a static web site. The rule will, on a match, produce the following alert:

```
[Thu Dec 03 20:02:50 2009] [error] [client 127.0.0.1] ModSecurity: …
Warning. Match of "rx ^(GET|HEAD)$" against "REQUEST_METHOD" required.…
[file "/home/ivanr/apache/conf/httpd.conf"] [line "464"]…
[hostname "192.168.3.100"] [uri "/"] [unique_id "SxgbacCoA2QAABC7HMgAAAAB"]
```

Alert messages contain quite a lot of information by default, but they do not provide enough. For example, the default message generated by ModSecurity gives you some idea about what the rule looks like, but it doesn't tell you what the rule writer wanted to accomplish. This is where meta data actions come into play. They are primarily used to document rules and make them easier to handle. Here is the same rule as above, but with additional meta data:

```
SecRule REQUEST_METHOD "!^(GET|HEAD)$" \
    "phase:1,t:none,log,block,id:1001,rev:2,\
    severity:WARNING,msg:'Request method is not allowed'"
```

I have added four meta data actions:

- The `id` action assigns a unique identifier to the rule, making it possible to match an alert to the rule that caused it. The addition of the ID also makes it possible for the rule to be manipulated, either at configuration-time or at run-time. The IDs are very important because they allow rule sets to be customised while leaving the original

configuration files intact (for example, using the SecRuleUpdateById and SecRuleRemoveById directives), which, in turn, allows for the automated upgrades of rule sets.

- The rev action (short for revision) is essentially a change counter, or a serial number: it starts at one and increments by one every time a rule changes. The idea is to make it possible to determine, at a glance, if a rule changed and, even better, to make it possible for a program (who wouldn't be able to understand the differences between two rule versions anyway) to do that.
- The severity action tells you how serious a detected problem is. ModSecurity adopted the *syslog* system of severities, which are listed in Table 14.1, "Severity values". The least serious severity is DEBUG (7) and the most serious one is EMERGENCY (1). That aside, there are no clear guidelines how to assign severities to rules, leaving each author to adopt his or her own system.
- Finally, the msg action adds another message to the rule, which should explain the goal of a rule, or its result.

The information in meta data actions is always used in alerts. The improved rule produces the following alert:

```
[Thu Dec 03 20:11:25 2009] [error] [client 127.0.0.1] ModSecurity: Warning.…
Match of "rx ^(GET|HEAD)$" against "REQUEST_METHOD" required.…
[file "/home/ivanr/apache/conf/httpd.conf"] [line "465"] [id "1001"] [rev "2"]…
[msg "Request method is not allowed"] [severity "EMERGENCY"]…
[hostname "192.168.3.100"] [uri "/"] [unique_id "SxgZZsCoA2QAABCYL9IAAAAA"]
```

That's much better, but the alert still does not explain why we do not allow any request method other than GET or HEAD. Let's try again:

```
# Do not allow request methods other than GET or HEAD. Allow
# site does not currently use any other methods; restricting
# the methods allowed reduces the attack surface.
SecRule REQUEST_METHOD "!^(GET|HEAD)$" \
    "phase:1,t:none,log,block,id:1001,rev:2,\
    severity:WARNING,msg:'Request method is not allowed because \
it is not used by the application',tag:HARDENING"
```

This latest batch of improvements added a long description of the rule functionality and also improved the alert message. In addition, I also used the tag action to categorise the rule. Tags are pieces of text that can be attached to rules. It is possible to attach one ore more tags. By convention, the first tag defines rule's primary category, and all other tags define secondary categories. Knowing a category for a rule is very useful to understand what the rule does. Categories also enable monitoring systems that collect alerts to construct pretty alert pie chars with little effort (e.g., displaying how many alerts of each category occurred in a time period). For tags too there are no clear guidelines how to use them. The Core Rule

Set does use them to categorize rules, but it does not document the categories (and does not guarantee that the categories won't change).

7 Rule Configuration

[...]

Apache Configuration Syntax

In the first instance, you should view Apache configuration as single file that consists of many lines of text. In reality, any configuration can be split among many files, but that's only for our convenience. To Apache, it's just line after line after line.

If you take a look at a typical configuration file, you will find that every line falls into one of three groups:

Empty lines
: Empty lines (either those that are genuinely empty, or those that contain only whitespace characters) have no function as far as Apache is concerned, but they help us make configuration files easier to read.

Comment lines
: Comment lines have the # character as the first non-whitespace character; any text can follow. Comment lines are often used to make configuration files user-friendly, providing documentation. They are also used to deactivate parts of configuration without deletion, which is handy if you ever want to put the deactivated parts into use again.

Data lines
: If a line is neither empty nor a comment line, it is a data line, and Apache will use it in configuration building.

Now, although the above three-group categorisation is very handy to use to organise things in your head, the truth is that you can have a comment on any line, even on a line that contains configuration data. Whenever Apache sees the # character on a line it will ignore it

and everything else that follows on the same line. However, that style of commenting is not widely used because it makes the configuration files more difficult to read.

With all this in mind, let's have a look at an example configuration fragment:

```
# It's always useful to begin configuration with a comment.
# Perhaps you have something important to say, for
# example, what is this configuration for?

# The one empty line above helps separate one comment from another.

# What follows is a single data line.
SecRuleEngine On

# What follows is a data line that also has a comment.
SecAuditEngine RelevantOnly # Only log interesting events.
```

Breaking lines

In practice, configuration lines can be as long as you need them to be. I am pretty sure there is a limit, but I've never encountered it. You will want your lines to be on the short side anyway. Most configuration tweaking and maintenance takes place remotely, so for best results your lines need to fit within your shell window. Otherwise you'll have to do a lot of scrolling or use the automated word-wrapping facility, if your editor supports it.

To split a long line into two you use a single backslash character followed by a newline:

```
SecRule ARGS KEYWORD \
    phase:1,t:none,block
```

Apache will interpret the above two-line configuration snippet as a single line. You can use this trick as many times as you wish, creating single logical lines that consist of multiple actual lines.

You can place a break at any location, but some places are better than others. I prefer to indent continued lines by 4 characters, but although my eye does not see the indentation, the 4 characters actually end up in the line. Unless you break the line in a place where whitespace does not matter, you will end up with a gap somewhere. The best place for a continuation is between directive parameters, like in the above example. With rules, the first two parameters are generally short, so in most cases you will place the continuation after the second parameter, again like in the above example. The third parameter, action lists, is often too long to fit even on a broken line that I often find myself breaking the parameter across lines. When you do that, the best place for a break is just after a comma (that's where whitespace does not matter).

Directives and parameters

Every data line begins with a directive name, followed by zero or more parameters. Apache supports the following directive parameter styles:

- No parameters.
- A single boolean parameter, which only allows for `On` or `Off` values (e.g., `SecRuleInheritance`).
- One, two or three free-form parameters, where each parameter has a separate meaning; parameters other than the first one can be optional (e.g., `SecRule` and `SecRuleScript`).
- Any number of free-form parameters, but all must have the same meaning (e.g., `SecResponseBodyMimeType`).

Directive parameter values are separated one from another using whitespace.

```
SecRule ARGS script
```

Exceptionally, if you have a value that contains one or more whitespace characters, you will have to enclose the entire value in question marks, a signal that will enable Apache to understand that there's only one parameter inside.

```
SecRule RESPONSE_BODY "Error has occurred"
```

When there are no whitespace characters inside parameter values you don't have to use question marks (even when the value contains a lot of unusual characters), but you can. Whatever you do, just be consistent and always use the same approach.

Spreading configuration across files

As your configuration keeps growing, you will find it more difficult to find your way around. That is especially true with ModSecurity, because not only will you have the configuration but there'll be many rules, some of which you may be writing yourself and some of which you may be downloading from an external source.

Apache configuration always begins with a single file, but you are allowed to include other configuration files using the `Include` directive. The following, for example, could be a skeleton for your ModSecurity configuration:

```
Include conf/modsecurity/main.conf
Include conf/modsecurity/preamble.conf
Include conf/modsecurity/rules1.conf
Include conf/modsecurity/rules2.conf
Include conf/modsecurity/rules3.conf
```

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

The paths I used in the above example are all relative; Apache will resolve using its main installation path (e.g., `/usr/local/apache`) as a starting point. Of course, you can use absolute paths if you wish, but that usually means more typing.

The `Include` directive can also include several files in one go, when you use the Unix shell-style wildcard characters. They are:

- `?` - any one character
- `*` - zero or more characters
- `\` - escapes the character that follows
- `[]` - exactly one character from the range (e.g., `[0-9]` for a digit)

The most common way to use this feature is to include all files that end with a particular suffix:

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

If a an `Include` line resolves to multiple files, they will be included in alphabetical order. That's quite logical, but does not always work as desired, because we tend to choose names based on the purpose the files serve. A common strategy is to use numbers in file names to control the order in which they are included. The above include line, used with the previously discussed hypothetical ModSecurity configuration, wouldn't include the files in the correct order. But the inclusion will be done in the correct order if we rename the files to the following ones:

```
00-main.conf
10-preamble.conf
20-rules1.conf
30-rules2.conf
40-rules3.conf
90-epilogue.conf
```

I have intentionally selected the larger range than needed (0–99) and left gaps between numbers, because that will allow me to insert new files in between the existing ones.

> **Note**
>
> If you point `Include` to a directory it will include all files in it, as well as all the files in all the subdirectories. This particular feature is not very useful because you will virtually never have a directory that will contain just the configuration files; there'll always be something else, and that something will eventually break your configuration. If your text editor automatically creates backup files, you may not

> get an error when a backup file is included, but your configuration may fail in unexpected ways.

Container directives

Apache supports two directive types. The standard variant, which you have already seen, is defined by a single configuration line (which may be split across several physical lines). The other variant, container directives, use a syntax similar to XML:

- They always come in pairs, which we call tags
- The starting tag begins with < and ends with >
- When it comes to parameters, the starting tag uses the same format as all other directives
- The ending tag begins with </ and ends >
- The ending tag cannot have parameters
- The directives enclosed in the pair of tags (including, possibly, other container directives) are nested in a new configuration context

Have a look at the following example:

```
# This is the main configuration context

<VirtualHost demo1.example.com>
    # This is the configuration context
    # used by demo1.example.com

    <Location /special/>
        # This is the configuration context
        # used by demo1.example.com/example
    </Location>
</VirtualHost>

<VirtualHost demo2.example.com>
    # This is the configuration context
    # used by demo2.example.com
</VirtualHost>
```

The main configuration context exists in every configuration. There are two further `VirtualHost` contexts, nested in the main configuration context, and one `Location` context, nested in one of the virtual hosts.

ModSecurity does not define any container directives itself (modules are allowed to create such directives too), but it relies heavily on all the container directives used by Apache.

Configuration contexts

Apache allows for several types of configuration context using container directives. Configuration contexts are a mechanism that allows you to apply configuration only to parts of the server. The example in the previous section already demonstrated the three most commonly used configuration contexts, the main configuration context, `VirtualHost` and `Location`, but there are others. Below is the complete list:

Main
: The main (implicit) configuration context is used by default. Unless a configuration uses explicit configuration contexts, the entire server will use the single configuration context.

VirtualHost
: The `VirtualHost` configuration context is used to create a new virtual host, possibly using configuration unique to it. Apache will automatically choose the correct virtual host to use for a request, based on the host information supplied in every request.

Location and LocationMatch
: The `Location` and `LocationMatch` directives both create a location-specific configuration context. Apache will automatically choose the correct location configuration context to use, based on the active virtual host and the information provided in every request's URI.

Directory and DirectoryMatch
: The `Directory` and `DirectoryMatch` directives both create a directory-specific configuration context. This type of context only makes sense when in a non-proxy deployment of Apache because proxies typically don't interact with the local filesystems. This type of context will be used only Apache determines which file on the local filesystem will be used to serve a request.

Files and FilesMatch
: The `Files` and `FilesMatch` directives both create a file-specific configuration context. Apache automatically chooses the correct file-specific configuration context to serve a request, but only after it determines which file on the local filesystem is to be used.

> **Note**
>
> There is practically no difference between the `Location`, `Directory` and `Files` directives and their respective `LocationMatch`, `DirectoryMatch` and `FilesMatch` counterparts. They each provide a different way to achieve the same effect. You should invest some time into studying the Apache documentation to understand how and why these directives are different.

Configuration merging

When configuration is simple, a request will only use one configuration context, but when configuration is complex, configuration context may overlap. For example, you may define some rules for a specific virtual host and some further rules for a specific location. Those two configuration contexts have to be merged into a single configuration context before a request that triggers both of them can be handled. Merging always takes place between two contexts at one time. Multiple merging operations will be performed when there are three or more configuration contexts to merge.

There are two aspects to understanding merging:

1. The parent-child relationship is significant, as is the order in which contexts are merged. For example, if you define a setting in both contexts, one of the two values may be overwritten by the other one. If there are three contexts to merged, with a different value for the same setting in each context, you need to understand the order in which merging operations will happen.
2. Apache initiates the process, but every individual module handles the merging of its configuration. Thus, to understand merging, you need to study the documentation of each module you are using. Some simpler modules may not support merging at all, while complex modules (e.g., ModSecurity) will use different merging strategies for different configuration directives.

The order in which contexts are merged can be quite complex to understand if you want to use every possible combination, but my advice is to simplify:

1. Use only the `VirtualHost` and `Location` container directives.
2. Remember that multiple `Location` containers (in the same virtual host) are processed in the order in which they appear in the configuration file.

If you follow my advice, your average configuration will always start with the one defined in the main configuration context, which will then be overwritten by the per-virtual host configuration, which will then be overwritten by per-location configuration.

Configuration Inheritance

ModSecurity uses two inheritance (configuration merging) strategies. The first strategy is used for the non-rule directives (e.g., SecRuleEngine) while the second applies to the rules.

Configuration inheritance

When it comes to the configuration settings, ModSecurity implements a straightforward merging strategy:

1. Child context inherits all configuration settings from the parent configuration context.
2. The settings explicitly defined in the child context will overwrite those defined in the parent context.

Consider the following example:

```
SecRuleEngine On
SecAuditEngine RelevantOnly

<VirtualHost www.example.com>
    SecRuleEngine DetectionOnly
</VirtualHost>
```

The configuration of the main context will always be easy to understand because there's no merging to take into account: what you see in the file is what you get.

```
SecRuleEngine On
SecAuditEngine RelevantOnly
```

As for the configuration of the one virtual host, you can work it out quickly using the two previously mentioned rules. Start with the configuration of the parent configuration context, and use `DetectionOnly` instead of the inherited `On`.

Rule inheritance

Because rules cannot overwrite one another in the way predefined settings can, different merging rules apply to them:

1. Child context inherits the rules from the parent context.
2. The rules defined in a child context are added after the rules defined in the parent context.

This, too, should be intuitive. For example:

```
SecRule ARGS K1 id:1001

<VirtualHost www.example.com>
    SecRule ARGS K2 id:1002
</VirtualHost>
```

In the above example, there will be one rule defined in the main configuration context (the rule 1001), but two in the virtual host (1001 first, then 1002). You should also remember that the placement of a child context within the parent context does not influence the configuration of either context. The following segment, although slightly different, arrives at the same configuration as the previous example:

```
<VirtualHost www.example.com>
    SecRule ARGS K2 id:1002
</VirtualHost>

SecRule ARGS K1 id:1001
```

This is because, in Apache, configuration processing is a 2-step process: configuration contexts are created in the first step, with merging following in the second. From that point of view, the two previous configuration snippets are practically identical.

Rule inheritance is a desired feature in most circumstances because you will generally specify your configuration in the main configuration context or in the virtual host container, and then use more specific per-location contexts for tweaking. In such circumstances, it makes sense to begin with the rules specified in the parent configuration context. If you ever find yourself in a need to completely redefine the rules that run in a specific location, ModSecurity allows you to disable rune inheritance using the `SecRuleInheritance` directive.

```
SecRule ARGS K1 id:1001

<VirtualHost www.example.com>
    SecRuleInheritance Off
    SecRule ARGS K2 id:1002
</VirtualHost>
```

In the above example there virtual host context, the configuration will contain only the rule 1002 because rule inheritance was disabled.

SecDefaultAction inheritance anomaly

There is one exception to the configuration merging rules outlined in the previous sections: the `SecDefaultAction` setting is not inherited. The exception is more a bug than anything else, and can lead to some very subtle problems and unexpected behaviour. For example:

```
SecDefaultAction phase:2,log,auditlog,deny
SecRule ARGS K1 id:1001

<VirtualHost www.example.com>
    SecRule ARGS K2 id:1002
</VirtualHost>
```

In the above example, the first line of the configuration will change the built-in default action list to activate blocking. The change will be picked up by the rule 1001, which follows in the same configuration context. The rule 1001 will thus block. In the nested configuration context for the `www.example.com` virtual host, because there is no inheritance of `SecDefaultAction`, the default action list will "revert" to the built-in value (`phase:2,log,auditlog,pass` in ModSecurity 2.6.11). The rule 1002 will thus only warn, although it would be more intuitive if it would block.

Rule Manipulation

When you write your own rules then it is logical to change them directly whenever you want to make a change. There are cases where changing the rules directly is not desired. For example, changing a third-party rule set effectively creates a fork and makes upgrades difficult. ModSecurity has a mechanism or two you can use to change rules without actually changing them at their original location. Instead, you are either changing the rules after they are loaded, at configure-time, or as transactions are evaluated, at run-time.

Whenever possible you should choose configure-time manipulation, as this approach results with best performance. On the other hand, configure-time manipulation is quite limited, because it is unconditional; it results with a permanent modification of the rule within a context. Run-time manipulation is slower, but flexible: with it you can use the rule language to evaluate a transaction in any way you choose and then make your modifications.

Removing rules at configure-time

ModSecurity supports a configure-time mechanism that allows the remove a rule whose ID you know to be removed. Alternatively, you can also remove the rule whose message you know. That is achieved using `SecRuleRemoveById` and `SecRuleRemoveByMsg`, respectively. The example below demonstrates both directives:

```
SecRule ARGS K1 log,deny,id:123
SecRuleRemoveById 123

SecRule ARGS K2 "log,deny,msg:'Strange error occurred'"
SecRuleRemoveByMsg "Strange error occurred"
```

The `SecRuleRemoveById` is quite flexible because it allows you to list any number of rule IDs and rule ranges (e.g., 123–129), and it will remove all the rules that match. The `SecRuleRemoveByMsg` directive is similar in flexibility because its one parameter is a regular expression and also supports removing multiple rules at once.

Removing rules at configuration time as presented in the examples above of course does not make any sense. But it will once I change the above example to be slightly different. Imagine that you have a third-party rule set that you want to use. Like this:

```
Include /opt/modsecurity/etc/thirdPartyRules.conf
```

When you deploy the rule set you discover that there's one rule that produces a high volume of false positives. You are now faced with a dilemma: do you remove the offending rule or do you live with it? If you do the former you will be forced to assume the maintenance of the rule set and you won't be able to automatically update it. If you do the latter you will have to tolerate the false positives.

But, armed with `SecRuleRemoveById` and `SecRuleRemoveByMsg` and the IDs (or messages) extracted from the false positives, you can now remove the offending rule without actually modifying the third-party rule set:

```
Include /opt/modsecurity/etc/thirdPartyRules.conf
SecRuleRemoveById 123
```

Thus, we've established that removing rules at configuration-time can be very useful if you are unable, for some reason, to modify the original rule sets. You will find another application for this technique if you ever need to customise your rule sets for parts of application, which is done by creating a more-specific configuration context in Apache:

```
<VirtualHost www.example.com>
    # Your ModSecurity configuration directives and rules here
    # ...

    # A more-specific configuration context in which
    # you don't want to run the rule 123
    <Location /moreSpecific/>
        SecRuleRemoveById 123
    </Location>
</VirtualHost>
```

Updating rules at configure-time

Speaking of changing rules at run-time, sometimes you'll encounter a rule that is not a false positive, but that just does something that you don't want. For example, there may be a rule that was hard-coded to block in a particular way, but you want to warn, or block in another way. You can change what rule does on a match, at run-time, using the `SecRuleUpdateActionById` directive.

```
SecRule ARGS K1 log,deny,id:123
```

```
SecRuleUpdateActionById 123 pass
```

For simplicity, the previous example showed two rules in the same configuration context but, as discussed in the previous section, changing rule actions is only useful in the cases when you can't change the rules, or when you don't want to.

Excluding rules at run-time

Armed with one or more rule IDs (or rule ID ranges), a rule that runs first can prevent other rules from running. Like in the following example:

```
SecRule ARGS K1 nolog,pass,ctl:ruleRemoveById=123
# ...other rules here
SecRule ARGS K2 log,deny,id:123
```

If the first rule matches the associated `ctl` action runs. Because the `ctl` action specifies `ruleRemoveById` with `123` as parameter, the engine will make a note that it should not run the rule `123`. Later on in the phase, if the engine reaches the rule `123`, it will skip over it.

8 Persistent Storage

This chapter is about the persistent storage mechanism of ModSecurity, which allows it to remember things. Without persistent storage you are condemned to look at only one transaction at a time, and wonder what came before it, and if what came before it is important for what you are going to do with it. With persistent storage, or memory, you are able to construct complex data models that contain the structures equivalent the structures used in applications. Those structures can, for example, represent IP addresses, sessions, and users. The truth is that you will be given a set of tools and techniques that were designed to be used together, but don't prescribe the exact scenarios how they need to be used. You will be able to come up with the scenarios yourself, and implement them as needed. The power is in the flexibility, as is with most of else what ModSecurity has to offer.

The persistent storage mechanism in ModSecurity can be described as a free-form database of sorts. You can have any number of tables, and within each table you can have any number of records. The beautiful things about those records is that you don't have to know in advance what they will contain. You are free to just add and remove information from them as you go. Somewhat paradoxically, the storage mechanism was designed with ultimately transient data in mind, so each record has an expiry mechanism built-in, which allows the database to essentially keep itself in shape, removing obsolete records on the fly.

That's all fine, I hear you say, but what is that we can use that database for? Here are a couple of things that you can do, and that I will show you how to do in the remainder of this chapter:

- Track IP address activity, attack and anomaly scores
- Track session activity, attack and anomaly scores
- Track user behavior over a long period of time
- Monitor for session hijacking
- Enforce session inactivity timeouts and absolute life span
- Implement periodic alerting
- Detect denial of service and brute force attacks

On top of the above, I am sure that you will find plenty of other scenarios in your own environment.

Manipulating Collection Records

In this section I will cover the basics of collection manipulation. In most cases the creation is all you need to do and ModSecurity will take care of everything else. The rest of the section will tell you the details you need to know when you want full control of the persistent storage mechanism.

Creating records

Creating a record is a matter of deciding on a key and invoking the `initcol` action. The `IP` collection, for example, is almost always initialized unconditionally using the remote IP address (with the help of variable expansion). Because the `REMOTE_ADDR` variable is always available, it is appropriate to initialize the `IP` collection early, in phase 1:

```
# Track IP addresses
SecAction phase:1,nolog,pass,initcol:IP=%{REMOTE_ADDR}
```

> **Note**
>
> A collection can be initialized with a record only once per transaction. If there are multiple invocations of the `initcol` action for the same collection (`IP` in the example above), the first invocation will be processed and all subsequent invocations will be ignored.

Although most collections use single variables for their keys, it's perfectly possible to create a key out of two or more variables. For example, sometimes you may get a large number of users behind the same IP address but you may still want to attempt to track them individually. Although there's not a way to do that reliably, a more granular way would be to generate record keys using a combination of IP address and a hash of the `User-Agent` field:

```
# Generate a readable hash out of the User-Agent
# request header and store it in TX.uahash
SecRule REQUEST_HEADERS:User-Agent ^(.+)$ \
    phase:1,pass,t:none,t:sha1,t:hexEncode,capture,setvar:tx.uahash=%{TX.0}

# Initialize the IP collection using a
# combination of IP address and User-Agent hash
SecAction phase:1,nolog,pass,initcol:IP=%{REMOTE_ADDR}_%{TX.uahash}
```

In ModSecurity 2.5.x you are allowed to create only the predefined collections listed in Table 8.1, "Allowed collections in ModSecurity 2.5.x". A future version of ModSecurity may allow you to use any name (that does not clash with the built-in variables).

Table 8.1. Allowed collections in ModSecurity 2.5.x

Collection	Create with	Description
GLOBAL	initcol	Global (per-server) data store
IP	initcol	Per-IP address data store
RESOURCE	initcol	Per-resource (URL) data store
SESSION	setsid	Per-session data store
USER	setuid	Per-user data store

The collection names are chosen to give clue to the intended usage and I trust you won't have any difficulty figuring out what it is. In any case, the rest of this section will show you how to use each of the collections.

You will notice that not all collections can be created using the `initcol` action. The `SESSION` and `USER` collections have a special initialization action each in order to support application namespaces (described in the section called "Application namespaces").

> **Note**
>
> There is no practical difference between creating a record and retrieving an existing record. The `initcol` action will automatically create a new record if one does not already exist.

Application namespaces

A single server running ModSecurity can serve many different sites with their own separate session IDs and user accounts. While the session IDs will overlap only very rarely (assuming the ID generation algorithm is solid), there's a good chance that the username collisions will be quite frequent. For example, I imagine that every other application uses `admin` for the main administration account.

ModSecurity uses *application namespaces* to deal with this problem, whereby you are able to manually specify application boundaries. Each application then receives a private space for its `SESSION` and `USER` collections, preventing overlaps. Applications are defined using the `SecWebAppId` directive. Your goal should be to use one unique application ID per application.

For example:

```
<VirtualHost www.ssllabs.com>
```

```
    SecWebAppId ssllabs
</VirtualHost>

<VirtualHost www.feistyduck.com>
    SecWebAppId feistyduck
</VirtualHost>
```

The method in which application namespaces are implemented is very simple. For normal collections, the collection name is used to name the SDBM file in which its data will be stored. For namespace-aware collections, the namespace is part of the name. Assuming the configuration as in the above example, the data persistence directory may contain the following files:

```
default_SESSION.dir
default_SESSION.pag
feistyduck_SESSION.dir
feistyduck_SESSION.pag
IP.dir
IP.pag
ssllabs_SESSION.dir
ssllabs_SESSION.pag
```

You can see that there is one global database for the IP collection, but three databases for sessions: one each for `ssllabs` and `feistyduck` applications, and one (`default`) for all other applications together.

Initializing records

The transparent record creation makes it difficult to perform record initialization, which you will need if you're writing a complex rule. The special record variable `IS_NEW` can be used to determine is a record is brand new. The idea is to allow you to test if this variable is set, and perform the initialization if it is:

```
# Track IP addresses
SecAction phase:1,nolog,pass,initcol:IP=%{REMOTE_ADDR}

# Set the default reputation value for new IP records
SecRule IP:IS_NEW "@eq 1" \
    phase:1,nolog,pass,setvar:IP.reputation=100
```

Controlling record longevity

The number of records in a collection can grow very quickly, especially in the cases where you use one or more records per IP address and you have many users. To preserve space

and improve performance, you want your records to be deleted as soon as you don't need them any more, but no sooner.

The principal way to control the removal of records is through the *inactivity timeout mechanism* built into the collections subsystem. This mechanism ensures the removal of the records that are no longer updated. Its operation is straightforward:

1. An inactivity timeout value associated with every record.
2. Records are scheduled for deletion as soon as they are created.
3. Whenever a record is persisted, the expiry time is re-calculated using the current timeout value.
4. The default inactivity timeout value is 3600 seconds, but can be changed to anything else by setting the special `TIMEOUT` collection variable (e.g., `setvar:IP.TIMEOUT=300`).

It is best-practice to configure the desired inactivity timeout value in the same rule in which the initialization takes place. Choose the correct value depending on what your collection does. Use the following list as guidance:

- IP tracking - hours
- Session tracking - days
- User tracking - months

Deleting records

In most cases you will not need to delete collection records explicitly, because it's much better to configure the correct timeout period and let the garbage collection process deal with the records after they expire. There are currently two ways in which records are deleted:

- A special garbage collection process runs periodically to examine all records in all known collections (i.e., the collections that have been activated during the transaction using `initcol`, `setsid` or `setuid`). This process will remove all expired records.
- When an attempt to retrieve an expired record is made, the record is deleted and replaced with a new one.

It is possible to delete records explicitly, in a round-about sort of way: if you have previously retrieved the record you wish to delete, you can force the deletion by unsetting the special `KEY` collection variable:

```
# Delete record
SecAction phase:1,nolog,pass,setvar:!IP.KEY
```

Detecting very old records

Because the expiry time of a record can potentially be re-set indefinitely, it is not impossible to have a record survive for a very long time. Although ModSecurity won't complain about a record that is too old, it does record the creation time, making it possible to write a custom rule to inspect it. The rule language lacks the needed arithmetic operators to compare numerical values, but this Lua script works just fine:

```
function main()
    -- Retrieve CREATE_TIME of the current IP record
    local createTime = m.getvar("IP.CREATE_TIME");

    -- If the variable is available and if the record is older
    -- than 24 hours, report the problem back
    if ((createTime ~= nil) and (os.time() - createTime > 86400)) then
        -- Retrieve the record key, which will
        -- make the error message more useful
        local key = m.getvar("IP.KEY");
        -- Match
        return "IP record older than 24 hours (" ..
            (os.time() - createTime) .. "s): " .. key;
    end

    -- No match
    return nil;
end
```

To use the rule, place it in a file called `check_ip_create_time.lua`, and call it with:

```
# Check the CREATE_TIME of the IP collection
SecRuleScript check_ip_create_time.lua phase:5,log,pass
```

If you wish you can delete such old records (using the technique described in the previous section), use the following rule instead:

```
# Delete very old IP collection records
SecRuleScript check_ip_create_time.lua phase:5,nolog,pass,setvar:!IP.KEY
```

To learn more about writing rules in Lua, go to Chapter 11, *Writing Rules in Lua*.

Collection Variables

What makes collections beautiful is the fact that they allow you to store any variable, and on a whim. Once you initialize a collection (and thus obtain a record), you can use the `setvar`

action to create, modify and delete collection variables. This section covers three additional features that persistent collections support, but ordinary collections don't:

- Built-in variables, which give you insight into how the record is used
- Variable expiry, which allows you to remove (expire) a variable at some point in the future
- Variable value deprecation, which allows you to reduce the value of a variable over time.

Built-in variables

Every persistent collections contains certain built-in variables, as seen in Table 8.2, "Built-in collection variables". The use of these variables is explained throughout this section, but they are generally populated using the information provided by the underlying persistence mechanism, allowing you to understand how individual records are used.

Table 8.2. Built-in collection variables

Name	Access	Description
`CREATE_TIME`	Read-only	Record creation time, in seconds since January 1st, 1970.
`IS_NEW`	Read-only	Flag which is set on a record that is yet to be persisted for the first time.
`KEY`	Read/delete	Record key. Can be unset, in which case the record will be deleted.
`LAST_UPDATE_TIME`	Read-only	The last record update time, in seconds (as above).
`TIMEOUT`	Read/write	The current timeout value, which will be used to extend the life of the record on the next write. The timeout is initially set to 3600 seconds.
`UPDATE_COUNTER`	Read-only	Incremented every time a record is persisted.
`UPDATE_RATE`	Read-only	Record update rate, in requests per second. This value is calculated using the `CREATE_TIME`, `LAST_UPDATE_TIME` and `UPDATE_COUNTER` values.

Variable expiry

The variable expiry mechanism enables you to schedule a variable to be expired (unset) at some point in the future. It is a feature you can use whenever you want to execute an action that will remain active for a period of time (long after the HTTP transaction that initiated is gone).

The best example for this feature is IP address blocking. Assuming you have the IP collection initialized, IP address blocking requires two rules:

1. One rule that will decide when to block an IP address, and set the appropriate flag in the IP collection (let's use `IP.blocked`).

2. The second rule that will block transactions originating from the flagged IP addresses.

For example:

```
# Detect attack and install a persistent IP address block
SecRule ARGS attack "phase:2,log,block,setvar:IP.blocked,expirevar:IP.blocked=60,\
    msg:'Blocking IP address for 60s'"

# Enforce a persistent IP address block
SecRule IP:blocked "@eq 1" \
    "phase:2,block,msg:'Enforcing earlier IP address block'"
```

> **Note**
>
> If you want to blocking to remain active for a very long period of time, make sure that the IP collection timeout value is longer than the blocking period. If an IP collection record expires the block will expire with it.

Variable value depreciation

Variable expiry works well for the things that are black or white, right or wrong. But when you have shades of gray, you'll need to use variable value depreciation (action `deprecatevar`), which is designed to work with the variables that contain numerical values. When you employ depreciation, the numerical value of your choice is gradually reduced over time until it reaches zero. This mechanism is usually used to work with anomaly or attack scores.

> **Note**
>
> The `deprecatevar` action is implemented in a different way from `expirevar`. Whereas `expirevar` uses a "fire-and-forget approach", meaning it is executed only once and remains active it finishes the job, the `deprecatevar` action needs to be invoked continuously, on every request, for as long as you need the depreciation to remain active. The recommended approach for `expirevar` is to use it in the same rule that creates (or updates) a variable. The recommended approach for `deprecatevar` is to use it unconditionally (with `SecAction`) in phase 5.

The `deprecatevar` action takes two positive integer parameters, separated by a forward slash. The first number defines how much a value will be reduced at one time, while the second number defines the decrement duration. Together, the parameters define the speed of depreciation. In the following example the value of the `IP.score` variable will be reduced by 1 every 5 seconds:

```
SecAction phase:5,nolog,pass,deprecatevar:IP.score=1/5
```

The way in which you choose the numbers matters, because the reduction in value is made at the discrete intervals defined by the duration parameter. That means that, although both `1/5` and `60/300` will both result with the same variable value after 300 seconds, in the first case there would be 60 decrements of 1 at 5-second intervals, whereas in the second case you will get just one decrement of 60 after 300 seconds.

For a complete example using depreciation, consider the following implementation of IP address attack scoring:

```
# Increment IP address attack score with every attack
SecRule ARGS attack \
    phase:2,log,pass,setvar:IP.score=+1

# Block IP addresses whose attack score is greater than 10
SecRule IP:score "@gt 10" \
    "phase:2,log,block,msg:'IP address anomaly score over 10 (%{IP.score})'"

# Decrement attack score by 1 every 5 seconds
SecAction phase:5,nolog,pass,deprecatevar:IP.score=1/5
```

If you look at the debug log, you may find the following two lines for each variable being depreciated:

```
[9] Deprecating variable: IP.score=1/5
[4] Deprecated variable "IP.score" from 17 to 15 (10 seconds since last update).
```

As you would expect, depreciation does not occur when there is no change in the value. In that case, you would see the following message in the debug log:

```
[9] Not deprecating variable "IP.score" because the new value (1) is the same as …
the old one (1) (2 seconds since last update).
```

Implementation Details

Persistent storage in ModSecurity is implemented using the SDBM library, which is part of the Apache Portable Runtime (APR). SDBM was selected because it was already available (ModSecurity depends on APR anyway) and because it allows for the control of concurrent access. The latter is very important, because in ModSecurity we deal with potentially many concurrent transactions.

Retrieving records

Collection records are retrieved when the `initcol` action is encountered. Assuming the collection was not previously initialized, ModSecurity will look for the appropriate SDBM

database and fetch the record with the corresponding key. You can examine the process when you increase the debug log level to 9:

```
[9] Resolved macro %{REMOTE_ADDR} to: 192.168.3.1
[9] Read variable: name "__expire_KEY", value "1263975870".
[9] Read variable: name "KEY", value "192.168.3.1".
[9] Read variable: name "TIMEOUT", value "3600".
[9] Read variable: name "__key", value "192.168.3.1".
[9] Read variable: name "__name", value "IP".
[9] Read variable: name "CREATE_TIME", value "1263970741".
[9] Read variable: name "UPDATE_COUNTER", value "21".
[9] Read variable: name "counter", value "21".
[9] Read variable: name "LAST_UPDATE_TIME", value "1263972270".
[4] Retrieved collection (name "IP", key "192.168.3.1").
[9] Recorded original collection variable: IP.UPDATE_COUNTER = "21"
[4] Added collection "IP" to the list.
```

The first line is the clue as to what the key used was. Following will be one line for every variable retrieved from the database. You will notice that some variable names begin with two underscore characters. Those variables are internal to ModSecurity, and you can probably guess from their names what they do. The variables whose names begin with the `__expire_` prefix are created by the `expirevar` action to keep track of when individual variables need to be expired.

Storing a collection

All records initialized during a transaction will be persisted after the transaction completes. In the simper of the two cases, persistence will be a straightforward write to the database:

```
[9] Wrote variable: name "__expire_KEY", value "1263975870".
[9] Wrote variable: name "KEY", value "192.168.3.1".
[9] Wrote variable: name "TIMEOUT", value "3600".
[9] Wrote variable: name "__key", value "192.168.3.1".
[9] Wrote variable: name "__name", value "IP".
[9] Wrote variable: name "CREATE_TIME", value "1263970741".
[9] Wrote variable: name "UPDATE_COUNTER", value "22".
[9] Wrote variable: name "counter", value "22".
[9] Wrote variable: name "LAST_UPDATE_TIME", value "1263972270".
[4] Persisted collection (name "IP", key "192.168.3.1").
```

> **Note**
>
> If, while looking at your debug logs, you discover that an initialized collection is not being persisted, that's because nothing was changed in it. When there are no

> changes in the record the copy in storage will be identical to that in memory, so there is no need to perform the expensive write operation.

More often than not, however, persistence will be of a multi-step process, as follows:

1. Lock database
2. Retrieve (for the second time) the record
3. For every numerical value that was changed, calculate the delta
4. Update the numerical values in the record retrieved in step 2 by applying the calculated deltas
5. Write record to disk
6. Unlock database

The debug log will show something similar to the following (note the delta calculations in between the read and write operations):

```
[9] Re-retrieving collection prior to store: IP
[9] Read variable: name "__expire_KEY", value "1263975870".
[9] Read variable: name "KEY", value "192.168.3.1".
[9] Read variable: name "TIMEOUT", value "3600".
[9] Read variable: name "__key", value "192.168.3.1".
[9] Read variable: name "__name", value "IP".
[9] Read variable: name "CREATE_TIME", value "1263970741".
[9] Read variable: name "UPDATE_COUNTER", value "21".
[9] Read variable: name "counter", value "21".
[9] Read variable: name "LAST_UPDATE_TIME", value "1263972270".
[4] Retrieved collection (name "IP", key "192.168.3.1").
[9] Delta applied for IP.UPDATE_COUNTER 21->22 (1): 21 + (1) = 22 [22,2]
[9] Delta applied for IP.counter 21->22 (1): 21 + (1) = 22 [22,2]
[9] Wrote variable: name "__expire_KEY", value "1263975870".
[9] Wrote variable: name "KEY", value "192.168.3.1".
[9] Wrote variable: name "TIMEOUT", value "3600".
[9] Wrote variable: name "__key", value "192.168.3.1".
[9] Wrote variable: name "__name", value "IP".
[9] Wrote variable: name "CREATE_TIME", value "1263970741".
[9] Wrote variable: name "UPDATE_COUNTER", value "22".
[9] Wrote variable: name "counter", value "22".
[9] Wrote variable: name "LAST_UPDATE_TIME", value "1263972270".
[4] Persisted collection (name "IP", key "192.168.3.1").
```

The locking and the delta calculations are necessary in order to ensure the integrity of the persisted numerical values. Without them, multiple concurrent transactions would overwrite one another's values, and the numerical values would be incorrect. By remembering the changes, rather than absolute values, ModSecurity ensures that numerical values are always correctly persisted. Unfortunately, there is no way to ensure the integrity of non-nu-

merical values in the concurrent access scenario (not without severe performance degradation, that is). On the positive side, non-numerical values are typically used where the danger of concurrent access is low, and less often at that.

Record Limits

The SDBM library imposes an arbitrary limit of 1008 bytes on the combined size of key length and record length. If you break this limit, the persistence operation will fail and you'll get the following message in your logs:

```
[1] Failed to write to DBM file "/tmp/IP": Invalid argument
```

ModSecurity uses about 200 bytes for its needs (mostly the built-in collection variables), which means that you practically have about 800 bytes left. Although 800 bytes does not sound like much, it's enough in most situations because rules generally only use numerical values in persistent storage.

> **Note**
>
> If you are running out of space, avoid using very long keys. Keys are stored as three copies: two copies are used by ModSecurity and one copy by SDBM itself. If everything else fails, you can always resort to "brute force" and recompile the APR to increase the size limit to a much higher value.

As a rule of thumb, you should avoid to store anything user-controlled in persistent storage. For example, you might want to store the value of the `User-Agent` request header in a `SESSION` collections to check for possible session hijacking attacks, but that value can be up to 8190 bytes long (that's the Apache's default request header limit). In such situations, it is better to store a value derived from the `User-Agent` value, instead of the value itself.

Practically speaking, you can use the `md5` and `sha1` transformation functions, which will "condense" any input to a fixed length. Because the output of those two transformation functions is binary, it is a good idea to follow with the `hexEncode` transformation, which will make the value printable. Below is the rule from an earlier example, which takes the value of the User-Agent request header and transforms it into a value (stored in `TX.uahash`) that can be used with persistent storage:

```
SecRule REQUEST_HEADERS:User-Agent !^(.+)$ \
    phase:1,t:none,t:sha1,t:hexEncode,capture,setvar:tx.uahash=%{TX.0}
```

Applied Persistence

In this section I will apply the previously discussed techniques to several real-life problems:

- Periodic alerting
- Denial of service attack detection
- Brute force attack detection

Although the rules themselves are interesting, the way how the techniques are combined is what is really important, and what's going to take your rule-writing skills to a whole new level!

Periodic alerting

Periodic alerting is a technique useful in the cases when it is enough to see one alert about a particular situation, and when further events would only create clutter. You can implement periodic alerting to work once per IP address, session, URL, or even entire application. First you choose the collection you want to work with, and then you create a special flag whose presence will tell you that an alert needs to be suppressed.

The best case for periodic alerting can be made when you're dealing with problems that are not caused by an external factor, which typically happens with rules that perform *passive vulnerability scanning*. Such rules detect traces of vulnerabilities in output, and alert on them. They are quite handy because they can alert about problems before they are exploited. If passive scanning rules are stateless, they may cause far too many alerts because they will report a problem whenever they see it, which may happen very frequently on busy sites. If you are faced with such a problem, you will have probably seen the first couple of alerts, and, even if you are not doing anything to deal with the discovered issue, don't really want to be reminded about it. That annoyance can be solved by updating passive vulnerability scanning rules to alert only once, as I will not demonstrate.

To start with, here's a simple rule that PHP version leakage in the `X-Powered-By` response header, which is an information leakage problem:

```
SecRule RESPONSE_HEADERS:X-Powered-By !^$ \
    "phase:3,log,pass,msg:'X-Powered-By information leakage'"
```

PHP version leakage is a minor issue that is good to know about, but not on every single web server hit. It's a problem in the configuration of the PHP engine, which means we can use the `GLOBAL` collection, which operates on per-server basis. The idea is to create a special record (in the `GLOBAL` collection) just for one problem and use it to keep track of the previous activity.

The following rule will detect X-Powered-By information leakage, but only warn about the problem once every 60 seconds:

```
SecRule RESPONSE_HEADERS:X-Powered-By !^$ \
    "chain,phase:5,log,pass,id:1001,\
    msg:'X-Powered-By information leakage(%{TX.temp} hits since last alert)',\
    initcol:GLOBAL=id1001,\
    setvar:GLOBAL.id1001_counter=+1"
SecRule &GLOBAL:id1001_flag "@eq 0" \
    "setvar:GLOBAL.id1001_flag,\
    expirevar:GLOBAL.id1001_flag=60,\
    setvar:TX.temp=%{GLOBAL.id1001_counter},\
    setvar:GLOBAL.id1001_counter=0"
```

I will walk you through what the rule does:

1. The first rule checks if the problem exists by looking for a non-empty X-Powered-By response header.
2. Upon successful detection, two actions are carried out:
   a. A record in the GLOBAL collection is initialized, using the constant key 1. By performing the initialization only after a match, we enhance performance of the requests that do not have the leakage problem.
   b. The counter value is increased by one. Even if we don't alert on the problem, we keep track of how many violations there were.
3. The second rule, which, being part of the same chain, is tested only after the first rule matches, tests the GLOBAL.id1001_flag variable, which will tell us if we've alerted in the last period of time. The presence of the variable is a sign that we shouldn't alert (and you will see why in the next step). If the variable is not present the rule will match, and the following three actions will be carried out:
   a. The GLOBAL.id1001_flag variable will be created.
   b. The GLOBAL.id1001_flag will be set to expire 60 seconds in the future.
   c. The value of the GLOBAL.id1001_counter variable is preserved in the temporary variable TX.temp.
   d. The counter (GLOBAL.id1001_counter) is then re-set back to zero.
   e. Finally, the match of the second rule will cause the entire chain to match and create an alert. Note how the chain message makes use of the temporary variable TX.temp, which stored the earlier value of the counter (which we've reset since).

> **Note**
>
> If you don't need to track how many alerts were suppressed you should omit incrementing GLOBAL.id1001_counter, which will save you a write to disk for every

> suppressed alert (which could be a write for every request to your site, depending on the nature of the problem being detected).

Even with the above elaborate scheme to implement periodic alerting, it is possible to get more than a few alerts for a problem that occurs very often (e.g., on every request). This is because processing a request takes time, so it is entirely possible for two requests to execute so close to each other and both retrieve the `GLOBAL` record when the variable `GLOBAL.id1001_flag` is not set, in which case they may both alert. We are minimizing the chances of that happening by choosing phase 5 for the rule and using late initialization. Collections are persisted right after the rules in phase 5 complete, which means the window of opportunity for the collision is minimised.

If you need suppression to work per application script, use the `RESOURCE` collection. The following rule is identical the previous example, except that the collection initialization is slightly different:

```
SecRule RESPONSE_HEADERS:X-Powered-By !^$ \
    "chain,phase:5,log,pass,id:1002,\
    msg:'X-Powered-By information leakage(%{TX.temp} hits since last alert)',\
    initcol:RESOURCE=%{SCRIPT_FILENAME},\
    setvar:RESOURCE.id1002_counter=+1"
SecRule &RESOURCE:id1002_flag "@eq 0" \
    setvar:RESOURCE.id1002_flag,\
    expirevar:RESOURCE.id1002_flag=60,\
    setvar:TX.temp=%{RESOURCE.id1002_counter},\
    setvar:RESOURCE.id1002_counter=0
```

The idea with the `RESOURCE` collection is that it will give you access to record that is unique for the script that will be processing the request. When ModSecurity is embedded in a web server, initialize the `RESOURCE` collection in phase 2 using `SCRIPT_FILENAME` (which will maps to the actual script on disk, no matter what the request URI looks like). In a proxy situation, you can only use the `REQUEST_FILENAME` variable, but bear in mind that there are situations where a single script is used for an unlimited number of request URIs. A proxy cannot differentiate between `/index.php/1001` and `/index.php/1002` and sees them as two different request URIs. A web server would see them as only one script (when you use `SCRIPT_FILENAME`). Furthermore, it is possible that you will have two locations (e.g., `/index.php` in two different virtual hosts). To avoid the chances for collision, you should also use the current hostname as part of the key.

Denial of service attack detection

Generally speaking, reacting to denial of service attacks from within a web server is less than ideal. When the target of an attack is the web server itself (e.g., the attacker is trying to overwhelm by sending a large number of requests, or keep a large number of connections open), then, by the time a requests reaches the web server, it will have already cased the damage. Denial of service attacks based on brute force should be handled by the network layer, where you are able to minimize the attack impact.

When it comes to attacks against applications, that's another story, and you may actually find ModSecurity very useful. Application attacks rely on being able to send cheap requests (in terms of resources needed to send them) to applications which will use disproportionally more resources (CPU, I/O, and RAM) to process them. Any application function that performs intensive work is a good attack choice. For example, most simple database-backed sites exercise no control over how many database connections they open and are easy prey. Send more than a handful requests to such a site, and it will suddenly start to malfunction.

The simplest approach to detecting DoS attacks is to check the value in the `UPDATE_RATE` variable of a collection. But, because collections are only persisted when there's a change to record, you need to ensure that the collection you are using is written to on every request that matters. A simple way to do that is to increment a counter on every request. Here's an example using the `IP` collection:

```
SecAction phase:1,nolog,pass,setvar:IP.counter=+1
SecRule IP:UPDATE_RATE "@gt 10" \
    "phase:1,block,msg:'Request rate too high for IP address: %{IP.UPDATE_RATE}'"
```

I have one concern about the above approach, though: I don't like the fact that the `IP` collection is written to on every request. Unless you are already doing something with the `IP` collection, the constant updating of the collection will add to your overall resource consumption. That does not mean it's not going to work well, but it does mean that you need to watch it.

It is possible to improve the performance by focusing only on those requests that really matter. If you examine your access logs, chances are you will find that only a fraction of all requests are forwarded to the application, with the rest being requests for static resources, such as images, JavaScript and CSS files. Static files are delivered efficiently by the web server and you can probably avoid tracking them in ModSecurity. By amending the first rule in the above example to only increment on a non-static request (using an unreliable method of checking the file extension, which will be sufficiently good in this case), we increase the efficiency of our application DoS detection.

For example:

```
# Only increment the counter if the
# request is for a non-static resource
SecRule REQUEST_FILENAME "!\.(jpg|png|gif|js|css|ico)$" \
    phase:1,nolog,pass,setvar:IP.counter=+1
```

> **Note**
>
> The UPDATE_RATE value is calculated over the lifetime of a record. If you keep the records alive for a very long period of time, a spike of activity (which may or may not be a DoS) will not significantly change the overall rate. If you choose to rely on the UPDATE_RATE calculation in your rules, you also need to remember not to increase the TIMEOUT value too much.

Starting with ModSecurity 2.6, there is another way to detect a potential application denial of service attack. ModSecurity 2.6 implements the DURATION variable, using which you can discover how long a transaction has been running. The idea is to keep track of how much time the web server is spending, per IP address, session, or user.

The following example keeps track of the resources spent on every IP address:

```
# Block the IP addresses that use too
# much of the web server's time
SecRule IP.load "@gt 10000" \
    "phase:1,t:none,block,\
    msg:'IP address load too high: %{IP.load}'"

# Keep track of how much web server
# time is consumed by each IP address
SecAction "phase:5,nolog,pass,\
    setvar:IP.load=+%{DURATION},\
    deprecatevar:IP.load=250/1"
```

You mustn't forget to use the deprecatevar action to ensure that the load value goes down during the periods of inactivity. Otherwise, the load will keep increasing and the block will never drop. Please note that the values I used in the example are completely arbitrary. They are not likely to work on your sites. Use the trial and error approach until you arrive the values that work for you. Similarly, have in mind that a client's communication speed may impact the time he spends with a transaction. Excessively large pages may have skewed DURATION values. If you are buffering response bodies, I suggest that you move the tracking rule from phase 5 (which occurs after a transaction is complete) to phase 4 (which occurs just before a response body is sent).

Finally, if you get tired during at looking at the debug log during the testing of the above rules, consider writing a content injection rule (see Chapter 10, *Content Injection*, for more information) to append IP address update rate or load to the end of each site page. You can

even use such a rule in production if you make the appended content invisible by putting it inside a HTML comment. Then just view the HTML source whenever you need to find out the value.

Brute force attack detection

Brute-force attack detection is conceptually similar to the approach used to detect denial of service attacks. You keep track of the authentication failures and you react when you feel an attack is taking place. Performance-wise, brute force detection uses less resources, because the rules only have work to do only when authentication takes place.

To start with, you need to understand how authentication failure is manifested, because the condition will be different for every application. You do that by using the application, recording all traffic to the logging script, and performing both successful and unsuccessful authentication. Your goal is to write a rule that will trigger on a failure, but not on success.

Let's assume that we're dealing with an application that uses the URL `/login.php` for all authentication requests: on success, the application redirects the user to `/index.php`. On failure, the application redirects back to `/login.php`, asking the user to try again. Our brute-force attack detection rule could thus begin with:

```
<Location /login.php>
    # Check for authentication failure
    SecRule RESPONSE_HEADERS:Location ^/login.php \
        "phase:5,t:none,log,pass,msg:'Failed authentication'"
</Location>
```

Once we verify that works as expected, we can move to manage the counters. Let's start with the `IP` collection first. The following rule will keep a per-IP address counter and alert only after seeing 25 authentication attempts, when it will clear the counter and start over:

```
<Location /login.php>
    # Check for authentication failure, maintaining
    # a counter that keeps track of how many failures were
    SecRule RESPONSE_HEADERS:Location ^/login.php \
        "phase:5,chain,t:none,setvar:IP.bf_counter=+1,nolog,pass,\
        msg:'Multiple authentication failures from IP address'"
    SecRule IP:bf_counter "@gt 25" t:none,setvar:!IP.bf_counter
</Location>
```

What we really want to do is block access for a period of time when too many authentication attempts are seen. We can do that with an additional flag and a rule that checks for it:

```
<Location /login.php>
    # Enforce an existing IP address block
```

```
    SecRule IP:bf_block "@eq 1" "phase:2,block,\
        msg:'IP address blocked because of suspected brute-force attack'"

    # Check for authentication failure
    SecRule RESPONSE_HEADERS:Location ^/login.php \
        "phase:5,chain,t:none,nolog,pass, \
        msg:'Multiple authentication failures from IP address',\
        setvar:IP.bf_counter=+1"
    SecRule IP:bf_counter "@gt 25" t:none,\
        setvar:IP.bf_block,\
        setvar:!IP.bf_counter,\
        expirevar:IP.block=3600
</Location>
```

And there, we have our brute-force detection rules, which will block anyone who misbehaves for one hour. I will now proceed to implement another layer of brute-force attack defence, keeping track of the per-username authentication failures. This is possible, but with some restrictions as you shall soon see.

For the second layer of defense we need the place where to store the second counter, of which we need to keep track no matter which IP address is used for access. It is only natural to use the USER collection, which was designed for that sort of thing—keeping track of information on per-user basis:

```
<Location /login.php>
    # Enforce an existing IP address block
    SecRule IP:bf_block "@eq 1" \
        "phase:2,deny,\
        msg:'IP address blocked because of suspected brute-force attack'"

    # Retrieve the per-username record
    SecAction phase:2,nolog,pass,initcol:USER=%{ARGS.username}

    # Enforce an existing username block
    SecRule USER:bf_block "@eq 1" \
        "phase:2,deny,\
        msg:'Username blocked because of suspected brute-force attack'"

    # Check for authentication failure and increment counters
    SecRule RESPONSE_HEADERS:Location ^/login.php \
        "phase:5,t:none,nolog,pass,\
        setvar:IP.bf_counter=+1,\
        setvar:USER.bf_counter=+1"

    # Check for too many failures from a single IP address
    SecRule IP:bf_counter "@gt 25" \
        "phase:5,pass,t:none,\
```

```
        setvar:IP.bf_block,\
        setvar:!IP.bf_counter,\
        expirevar:IP.block=1800"

    # Check for too many failures for a single username
    SecRule USER:bf_counter "@gt 25" \
        "phase:5,t:none,pass,\
        setvar:USER.bf_block,\
        setvar:!USER.bf_counter,\
        expirevar:USER.block=1800"
</Location>
```

Session Management

Session management is one of the more fun aspects of ModSecurity, and an area where ModSecurity can be truly useful. The reason for that is simple: unlike with other methods, with sessions you practically get to understand and monitor what one single user does. The usefulness of session-tracking will vary depending on what you're protecting, but it's best used with applications that use sessions to enable the users to establish a "relationship" with the application. Because sessions are required to use an application in a meaningful way, the adversaries are compelled to use them too, and that makes monitoring easier.

Initializing Sessions

Before you start to think about session initialization, think about how many applications you have on the same server. If you have more than one you must follow the advice in the section called "Application namespaces", and create a separate application namespace using the `SecWebAppId` directive. Even if you have only one application, it doesn't hurt to use `SecWebAppId` because it causes the application ID to be recorded in audit logs. Over time you may add more applications, in which case it would be useful to know which audit log entries belong to which application.

To initialize a session, you need to know two things:

Extract session token from request
: Most applications use cookies to transmit session tokens. Session cookies' names vary, but they are always easy to identify because they will only contain a large, random-looking, string (for example, `64c24d4e35dc753cd085ca574def4131`). A small number of applications embed session tokens in their URLs, and they are even easier to identify because the large string can be seen in your browser's URL bar.

Configure sufficient session lifetime

ModSecurity collections have the default value of 3600 seconds, but that's too short for sessions, whose life span is measured in hours. Some faulty application might even not impose a limit of session duration. To be able to monitor sessions throughout their life, you need to choose a timeout value that is at least as long as the duration of the longest possible application session. In most cases you should aim the `SESSION` collection to remain alive for several times the maximum duration of the application session, because that will allow you to perform reliable session blocking. For the examples in this section I will use 48 hours (172,800 seconds) as the `SESSION` collection timeout value.

To initialize a session from a cookie, you first need to identify the correct cookie. Have a look at one request that contains session information:

```
GET /index.php HTTP/1.1
Host: 192.168.3.100:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-GB; rv:1.9.1.7) …
Gecko/20091221 Firefox/3.5.7 (.NET CLR 3.5.30729)
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=64c24d4e35dc753cd085ca574def4131
Pragma: no-cache
Cache-Control: no-cache
```

I've emphasized the session cookie, and you can see that it is very easy to identify the session token. In the effort to extract the session token, you won't have to deal with the request header directly. Because ModSecurity parses inbound cookies, you'll be able to retrieve it by name using the `REQUEST_COOKIES` variable. Session initialization is thus as simple as:

```
# Initialize SESSION from PHP session token
SecRule REQUEST_COOKIES:PHPSESSID !^$ \
    "phase:2,nolog,pass,\
    setsid:%{REQUEST_COOKIES.PHPSESSID},\
    setvar:SESSION.TIMEOUT=172800"
```

It is advisable, however, to verify session tokens before you use them as collection keys. Anything user-supplied should be validated first, because you never know what will get. Also, if the token is invalid then it will probably not be recognized by the application, in which case you probably don't have any reason to use it either.

In the following example, the first rule will initialize a session provided the session token is correct. The second rule, which will run only if the first one doesn't (notice `skip:1` in the first rule), will block a transaction with an invalid token.

```
# Initialize SESSION from PHP session token
SecRule REQUEST_COOKIES:PHPSESSID ^[0-9a-z]{32}$ \
    "phase:2,nolog,pass,skip:1,\
    setsid:%{REQUEST_COOKIES.PHPSESSID},\
    setvar:SESSION.TIMEOUT=172800"

# Catch invalid PHP session tokens
SecRule REQUEST_COOKIES:PHPSESSID !^[0-9a-z]{32}$ \
    "phase:2,log,block,msg:'Invalid session token'"
```

If your application uses URI-based session tokens, head to the section called "Capturing data", where I give a complete example how to use the data capture facility to extract session tokens from URIs.

Blocking Sessions

After the `SESSION` collection is initialized, blocking a session is a matter of setting a flag (with the correct expiry time) and checking for it on all requests. You have seen this technique earlier in this chapter. I describe the flag method, as well as several variations and other blocking methods in the the section called "Advanced Blocking".

Forcing Session Regeneration

Blocking sessions might work well for security but it isn't very user friendly. If you use session blocking alone you may leave your users confused, because they won't be able to continue to use the application and won't know how to obtain a new session (close all browser windows and start using the application again). The solution to that problem is to generate a new session for the user. There are two ways to achieve session regeneration, and I will demonstrate both here.

Both approaches use header manipulation, which means that you will need to use ModSecurity in tandem with `mod_headers`. (At this point you may wish to go to read the the section called "Integration with other Apache modules" first, where the I explain how to get ModSecurity to collaborate with other Apache modules.)

The following code contains two `mod_headers` rules, each activated by setting an environment variable:

```
# Neutralize the cookies containing disabled session IDs
```

```
RequestHeader edit Cookie "(?i)^(PHPSESSID)=(.+)$" "DISABLED_$1=$2" \
    env=DISABLE_INBOUND_SESSION

# Instruct browser to delete session cookie
Header always set Set-Cookie "PHPSESSID=;expires=Fri, 31-Dec-1999 00:00:00 GMT" \
    env=DISABLE_OUTBOUND_SESSION
```

The first rule is activated by the `DISABLE_INBOUND_SESSION` environment variable, after which it renames inbound session cookies. When a session cookie is renamed it is no longer a session cookie, but some cookie whose value will be ignored. As a result, the application will likely generate a brand new session cookie.

The second rule is activated by the `DISABLE_OUTBOUND_SESSION` environment variable, and sends a command to the user's browsers to delete the session cookie (by using the same name as the session cookies, with an expiry time in the past).

To maximize both security and usability, use both mechanisms in your rules: delete the session cookie of a session you are deciding to block (by executing `setenv:DISABLE_OUTBOUND_SESSION` in any phase except phase 5), and suppress inbound session cookies of the sessions that have previously been blocked (by executing `setenv:DISABLE_INBOUND_SESSION` in phase 1 or in phase 2).

Restricting Session Life Time

Because sessions in today's web applications function as temporary passwords, it is important to cancel them as soon as they are not needed. Two mechanisms are typically used to do that:

Inactivity timeout
: When a session is not used for a period of time it is reasonable to assume that it had been abandoned. Allowing such sessions to remain only increases the danger of them being re-used by someone who is not the original user.

Session duration timeout
: You also want to put an absolute limit on session duration. Very long session life span is very unusual and may be an indication automated activity, or a bad guy trying to extract as much information as possible from a hijacked session.

Here's what we need to do to implement the above two limits:

1. Record the last time a session is used. If you recall from earlier sections, whenever a collection record is persisted its `LAST_UPDATE_TIME` variable is updated. We need that value. So, in order to force session records to be persisted, we'll use the same ap-

proach as the one used with the IP collection— increment an arbitrary variable on every request.

2. Now that we have access to LAST_UPDATE_TIME, we can check it on every request to ensure that it hasn't been too long since the previous request.
3. All collections have the CREATE_TIME variable, which we'll use to enforce maximum session duration.

We'll use the following Lua rule (placed in the file check_session.lua) to check those two conditions:

```
function main()
    -- Retrieve session key
    local key = m.getvar("SESSION.KEY");

    -- If there's no key there's no session,
    -- so return without a match.
    if (key == nil) then
        return nil;
    end

    -- Retrieve CREATE_TIME
    local createTime = m.getvar("SESSION.CREATE_TIME");

    -- If the session was created more than 8
    -- hours ago, trigger a match
    if (os.time() - createTime > 28800) then
        -- Match
        return "Session older than 8 hours: " .. key;
    end

    -- Retrieve LAST_UPDATE_TIME
    local lastUpdateTime = m.getvar("SESSION.LAST_UPDATE_TIME");

    -- Check for a period of inactivity
    if (os.time() - lastUpdateTime > 600) then
        -- Match
        return "Session inactive for more than 10 minutes ("
            .. (os.time() - lastUpdateTime) .. "s):" .. key;
    end

    -- No match
    return nil;
end
```

Because this particular feature is more complex than your average rule, I am going to put all the required rules together in a self-contained example, which combines everything we've

discussed about session initialization, collection timeouts, session inactivity detection (the Lua rule), session blocking, and header manipulation:

```
# Initialize session
SecRule REQUEST_COOKIES:PHPSESSID ^[0-9a-z]{32}$ \
    "phase:2,nolog,pass,\
    skip:1,\
    setsid:%{REQUEST_COOKIES.PHPSESSID},\
    setvar:SESSION.TIMEOUT=172800"
SecRule REQUEST_COOKIES:PHPSESSID !^[0-9a-z]{32}$ \
    "phase:2,log,block,msg:'Invalid session token'"

# Check for expired session
SecRule SESSION:expired "@eq 1" \
    "phase:2,log,redirect:/session-timeout.html\
    setenv:DISABLE_INBOUND_SESSION,\
    setenv:DISABLE_OUTBOUND_SESSION"

# Check session inactivity and duration
SecRuleScript check_session.lua \
    "phase:2,log,redirect:/session-timeout.html,\
    setvar:SESSION.expired\
    setenv:DISABLE_INBOUND_SESSION,\
    setenv:DISABLE_OUTBOUND_SESSION"

# Increment the session counter
SecRule REQUEST_FILENAME "!\.(jpg|png|gif|js|css|ico)$" \
    phase:1,nolog,pass,setvar:SESSION.counter=+1

# Neutralize the cookies containing disabled session IDs
RequestHeader edit Cookie "(?i)^(PHPSESSID)=(.+)$" "DISABLED_$1=$2" \
    env=DISABLE_INBOUND_SESSION

# Instruct browser to delete the session cookie
Header always set Set-Cookie "PHPSESSID=;expires=Fri, 31-Dec-1999 00:00:00 GMT" \
    env=DISABLE_OUTBOUND_SESSION
```

Detecting Session Hijacking

Session hijacking is a potentially devastating attack, often executed as the next step after a successful XSS attack. Once the attacker obtains a session token, he can assume the identity of the original user. Although it is not possible to detect and prevent session hijacking 100% reliably, there are a few defenses that can prove to be very effective. Before you resort to stateful session monitoring as a measure against session hijacking, however, you should verify that have done everything you can to secure the session cookies—if you make them

safe from compromise then session hijacking is not possible. I discuss the necessary session cookie rewriting in the section called "Integration with other Apache modules".

Our session hijacking detection measures are going to focus on two pieces of information:

Session IP address

Sessions are not attached to IP addresses. Anyone with the knowledge of the session token is allowed to participate in a session. Having said that, the IP address to which the session was initially assigned (on the first request) will in many situations remain the same throughout a session. For example, a user accessing an application from his workstation attacked to the internal network is not likely to change his IP address. That's probably the best-case scenario.

When it comes to Internet users and roaming users, the change of IP address is possible and you can never be quite sure if a hijacking is taking place or not. For example, AOL is famous for their proxies using a number of completely different addresses. The rumour is that AOL users can have a different IP address on every subsequent request. Roaming users can start a session while they are in one place, put their laptop in standby, and resume the session from a completely different place.

It should also be said that it is possible for the attacker and the victim to have the same IP address as far as you are concerned. That could happen, for example, if they are behind the same proxy or a network address translation (NAT) system.

Ultimately, the value of this detection mechanism will depend on your user base. My advice is to try the mechanism out as a warning system initially, and see if it produces false positives.

Session User-Agent

Whereas it's possible that the session IP address will change, it's far less likely that the user agent identification will. If you start a session in one browser, it's very unlikely that you will finish it in another. Unless you hijack someone's session, that is. Research carried out by the Electronic Frontier Foundation [https://www.eff.org/deeplinks/2010/01/tracking-by-user-agent] indicated that one in about 1500 users have the same `User-Agent` request field. Checking that the user agent identification remains the same across all session requests is thus a decent detection mechanism. It is also a mechanism that can be easily defeated by an determined attacker who knows it exists and who can somehow uncover the victim's own identification string (with a bit of social engineering, for example).

Putting that lengthy discussion aside, here's how to store the original IP address and `User-Agent` values and check them on subsequent requests:

```
# Initialize session
```

```
SecRule REQUEST_COOKIES:PHPSESSID ^[0-9a-z]{32}$ \
    "phase:2,nolog,pass,\
    skip:1,\
    setsid:%{REQUEST_COOKIES.PHPSESSID},\
    setvar:SESSION.TIMEOUT=172800"

# Reject requests that use invalid session tokens
SecRule REQUEST_COOKIES:PHPSESSID !^[0-9a-z]{32}$ \
    "phase:2,log,block,msg:'Invalid session token'"

# Generate a readable hash out of the User-Agent
# request header and store it in TX.uahash
SecRule REQUEST_HEADERS:User-Agent ^(.+)$ \
    "phase:2,nolog,pass,t:none,t:sha1,t:hexEncode,capture,\
    setvar:TX.uahash=%{TX.0}"

# Initialize SESSION, storing a hash of the User-Agent
# value, as well as the originating IP address.
SecRule SESSION:IS_NEW "@eq 1" \
    "phase:2,nolog,pass,\
    skipAfter:AFTER_SESSION_HIJACKING_RULES,\
    setvar:SESSION.uahash=%{TX.uahash},\
    setvar:SESSION.ip=%{REMOTE_ADDR}"

SecRule SESSION:ip "!@streq %{REMOTE_ADDR}" \
    "phase:2,pass,msg:'Possible session hijacking: Expected session address …
%{SESSION.ip} but got %{REMOTE_ADDR}'"

SecRule SESSION:uahash "!@streq %{TX.uahash}" \
    "phase:2,pass,msg:'Possible session hijacking: Expected session U-A hash …
%{SESSION.uahash} but got %{TX.uahash}'"

SecMarker AFTER_SESSION_HIJACKING_RULES
```

There's nothing in the above rules that you haven't already seen—they are just a combination of the techniques already covered in this chapter.

User Management

When it comes to persistent state, user management is the final piece of the puzzle. By following individual users, you come as close as possible to using the same data model the applications do. I have already used the USER collection in this chapter to keep track of authentication attempts. Now we're going to see if it is possible to detect users as they sign in and out. (Of course it is.) If we manage to detect those two events, we might be able to associate each session with a user account and use that information to initialize the USER collection.

One thing to have in mind is that tracking users in ModSecurity is not going to be exact science. You have to work with the information you have available, which means that you are going to have to rely on many assumptions—some of which may not be true. That's going to be just fine for as long as you use the user management facilities with that unreliability in mind.

Detecting Users Sign In

The work we'll need to perform to detect a sign-in event is just the opposite of what we did to detect brute force attacks against authentication. I will base the examples in the section on the assumption that we're dealing with an application whose sign-in form is located at `/login.php`, and that the application redirects back to the home page (`/index.php`) when authentication is successful. (If you recall, in the case of failed authentication the redirection was back to the same `/login.php` page.)

The following example assumes the `SESSION` collection was initialized by an earlier rule:

```
# Initialize the USER collection based on
# the user information we keep in the session store
SecRule SESSION:user !^$ \
    "phase:2,nolog,pass,\
    setuid:%{SESSION.user},\
    setvar:USER.TIMEOUT=2592000"

<Location /login.php>
    # Check for successful authentication
    SecRule REQUEST_METHOD "@streq POST" \
        "phase:5,chain,t:none,nolog,pass"
    SecRule RESPONSE_HEADERS:Location ^/index.php \
        setvar:SESSION.user=%{ARGS.username}
</Location>
```

The first of the two rules simply looks in the `SESSION.user` variable to see if we had previously established who the user behind the session is. If the information is available, the rule uses it to initialize the `USER` collection. The second rule detects the sign-in event, as previously discussed.

> **Note**
>
> It is a good idea to write a set of positive security rules to whitelist the sign-in function. Look at the previous example carefully and try to answer what will happen if the sign-in function is invoked with two (different) `username` parameters in the request. Do you think that it's possible that the application uses one of those parameters where we use the other (and associate the session with the wrong user

> account)? It certainly is possible. But, if you write a couple of rules for the sign-in page to check that requests contain only the parameters you expect, and that the cardinality of each parameter is correct, you minimize the danger of doing the wrong thing.

Detecting Users Sign Out

Detecting the sign-out function is much easier, as the action is rarely conditional. In the following example, I assume an application where it is enough to visit the `/signout.php` page in order to sign out of the application:

```
<Location /signout.php>
    # Disassociate user from session
    SecAction phase:5,nolog,pass,setvar:!SESSION.user
</Location>
```

When we're on the sign-out page, we only need to remove the user information from the `SESSION` collection and we're done.

9 Practical Rule Writing

[...]

Whitelisting

Rule sets are usually written to single out unusual requests, but it turns out that most networks contain one or more sources of requests that are not only unusual but also desired. The more complex the network the more likely it is that you'll need to use whitelisting. In most cases it will be a crude monitoring script that watches your web site that will look most like a crude Perl script that is attacking it. In others, you might have outsourced security testing to a third party and you don't want your rules to interfere with their work. Finally, even if you don't have any of that, you won't be able to avoid the unexpected—Apache sending requests to itself.

Whitelisting theory

You have to be very careful when writing whitelisting rules, because each such addition to your rule set creates a bond of trust. If you make a mistake you can end up with a hole in your rule set that can be used by your adversaries. You will be asking yourself three questions:

How do I know the request is from the person or device I want to whitelist?
: In the ideal case the remote client will authenticate itself in some way, ideally with a password (embedded, for example, in the `User-Agent` request header). The drawback of this approach is that you will probably have to make some configuration changes on the remote end to support authentication, and sometimes that might not even be possible. As an alternative to authentication, you can choose to look at the IP address or range from which the request is coming. If you do, take a moment to consider if it possible (and how likely it is) that something else can send you (potentially malicious) requests from the same IP address range.

Is there anything specific about the requests I want to whitelist?
: You may have established that the requests are coming from a source you can reasonably trust, but it's still a good idea to narrow down the attack vector as much as possible. Observe, over time, the requests you want to whitelist: is there a recurring pattern? For example, most monitoring requests are identical. In other cases, the requests will restricted to a part of your web site and will have predictable parameters.

What changes do I want to make to the default configuration?
: The last question pertains to the action you wish to take after you definitely decide that you want to go through with whitelisting. The easiest thing to do is to simply use the allow action to let the remote party to continue unconditionally, but are you really comfortable giving them unrestricted access? A better solution might to be to switch the rule engine to detection mode. You will not regret this, for as long you get false positives only occassionally.

In the next section I will discuss the placement of whitelisting rules, followed by several simple examples and finishing with the rule you will need to silence the Apache web server itself.

Whitelisting mechanics

Whitelisting rules need to be executed before all your other detection rules, which means they should always follow your configuration and system rules. It is a good idea to have a special file for this category of rule alone. That will make them easy to find, after a simple glance at the list of your configuration files.

Most whitelisting rules will look at the remote address first, so let's do just that. Let's assume that there is a trusted employee to whom you want to give unrestricted access to your web site. The IP address of his workstation is 192.168.1.1. The whitelisting rule is as follows:

```
SecRule REMOTE_ADDR "@streq 192.168.1.1" \
    phase:1,t:none,nolog,allow
```

Because you only need to work with one IP address you use the `@streq` operator. Upon detecting a request from the employee's IP address, the allow action will interrupt the operation of the rule engine, skipping all phases except phase 5.

ModSecurity does not have an operator designed specifically to work with IP addresses, so you'll need to be creative if you are given several IP addresses to whitelist. Suppose you are given three IP addresses. You could write three seperate rules, but that is not only inelegant, but inefficient too. In most such cases, the `@rx` operator will do the job. For example:

```
SecRule REMOTE_ADDR "@rx ^192\.168\.1\.(1|5|10)$" \
    phase:1,t:none,nolog,allow
```

The above example whitelists three IP addresses: 192.168.1.1, 192.168.1.5 and 192.168.1.10.

As previously discussed, you should avoid using the allow action whenever you can. It is recommended to swith the rule engine to detection-only instead:

```
SecRule REMOTE_ADDR "@streq 192.168.1.1" \
    phase:1,t:none,nolog,pass,ctl:ruleEngine=DetectionOnly
```

In the above rule, I replaced `allow` with `pass` (which won't do anything else but move to the next rule once the current rule is done), and added an invocation of the `ctl` action with the instruction to change the operating mode of the rule engine.

If you want to take my advice and require some form of authentication in order to activate your whitelisting rules, consider the following example where I also require a correct password to be placed in the User-Agent request header:

```
SecRule REMOTE_ADDR "@streq 192.168.1.1" \
    phase:1,t:none,nolog,pass,ctl:ruleEngine=DetectionOnly,chain
SecRule REQUEST_HEADERS:User-Agent "@contains SECRET_PASSWORD"
```

Granular whitelisting

Although every invocation of the `allow` action interrupts the phase in which it runs, you are able to choose whether and how other phases in the same transaction are affected. The allow action has an optional parameter, and the following rules apply:

Interrupt current phase and skip all other inspection phases
: If you invoke `allow` without a parameter then, no matter of the current phase, all inspection phases will be skipped.

Interrupt current phase only
: When `allow` is invoked with `phase` as parameter (`allow:phase`), it restricts the effect of this action only to the current phase.

Interrupt current phase and any remaining request phase
: When `allow` is invoked with `request` as parameter (`allow:request`), as part of phase 1, it will be interrupted and the second request phase (phase 2) will be skipped. Phase processing will continue with the first response phase (phase 3).

Complete whitelisting example

Earlier in this section I mention how Apache talks to itself. Because that is something every ModSecurity administrator will have to deal with, I will use the use case to demonstrate the steps you should take in the development of a whitelisting rule.

First, let's look at the complete request we need to allow ignore:

```
::1 - - [26/Oct/2009:16:01:06 +0000] "OPTIONS * HTTP/1.0" 200 - …
"-" "Apache (internal dummy connection)"
```

What can we deduce from the above log line:

1. The first thing that you will notice about this request is that it always arrives from the server itself. In the example the remote address is ::1 (IPv6 localhost). In other cases you will see 127.0.0.1 there. We can use this information to restrict the source of requests that our rule will take into account.
2. The request is always the same and involved the `OPTIONS` request method. This is even more helpful because it allows us write a rule that only matches that specific usage.
3. The user agent identification is the same for all requests.

Using the obtained information, we write a robust and safe rule:

```
SecRule REQUEST_LINE "@eq OPTIONS * HTTP/1.0" \
    "phase:1,t:none,chain,allow,nolog"
SecRule REMOTE_ADDR "^(::1|127\.0\.0\.1)$" t:none
```

I have used only the first two facts for my rule because I have felt they allow me to uniquely identify a request that no more narrowing is needed. Besides, the User-Agent request header is trivial to subvert.

Can the above rule be improved? Sure it can:

1. We restrict the remote address and the request line, but there's no mention of request headers. In theory, someone could place exploit payload into one of the request headers, sending it using the `OPTIONS` request method. As an exercise, record one Apache request to the audit log, examine the request headers it sends, and modify the above rule to nail every header down.
2. The really paranoid could look in the Apache source code to change the default user agent identification and thus allow for reliable identification of Apache access.

Virtual Patching

[...]

Reputation Management

[Find a better title for this section. GeoIP. RBL. Private IP address and user reputation management.]

Organizing Rule Sets

[...]

Using Rule Sets

[...]

Integration with other Apache modules

One of the biggest advantages of Apache is its modular nature. When you put modularity and popularity together, it sometimes seem that, whatever need you can think of there's already a module to fulfill it. In most cases modules are used on their own, but multiple modules can sometimes communicate one with another. ModSecurity generally tries to avoid reimplementing the features available in other modules, even for the functionality that could come under the security label. Thus there will be times where you will need to send or receive instructions to and from other modules.

There are two mechanisms in Apache that allow for the communication among modules:

Environment variables
: Intra-module communication using environment variable is the common approach to having modules exchange information and influence one anther. Whenever two modules need to communicate, the receiving module will be configured to watch for the presence (and possibly the value) of a particular environment variable, and act on it. Many modules are built with environment variables in mind, so whenever you discover that a particular module supports it, that means you can use it to talk to the module from ModSecurity using the `setenv` action. Because in ModSecurity you can use the `ENV` collection to retrieve the value of a named variable, you can write rules that use the information prepared by other modules, without limitation.

Optional functions
: Optional functions make possible for a module to export one or more named functions for other modules to consume. This mechanism is intended for module devel-

opers to use and chances are you won't be using it very often. ModSecurity builds its extension APIs on top of optional functions. The extension APIs are described in Chapter 13, *Extending Rule Language*.

The modules you may find yourself integrating with are:

- `mod_deflate`
- `mod_headers`
- `mod_log_config`
- `mod_rewrite`
- `mod_setenvif`

Conditional logging

Normally, an access log will record every transaction processed by Apache, but sometimes you will want to record only some transactions. That is called *conditional logging*. Apache's logging facilities support it, enabling you to use environment variables to decide what to log.

- Log by default, but do not log if an environment variable is set
- Do not log by default, but log if an environment variable is set

In the following example, I create a custom access log that only logs the transactions from a specific IP address:

```
# Detect the condition that require logging
SecRule REMOTE_ADDR "@streq 192.168.1.1" \
    phase:1,nolog,pass,setenv:SPECIAL_ACCESS_LOG

# Create a special access log file, which reacts to
# the SPECIAL_ACCESS_LOG environment variable.
CustomLog logs/special_access.log combined env=SPECIAL_ACCESS_LOG
```

> **Note**
>
> With ModSecurity 2.5.x it is not possible to control logging with a phase 5 rule, because this phase executes only after Apache has completed writing to its log files. There is currently a discussion about moving phase 5 to execute just before Apache does its logging and ModSecurity 2.6.x probably won't have any restrictions in this respect.

Header manipulation

In Apache, the `mod_headers` module is used for header manipulation. Its `Header` and `RequestHeader` directives know how to look up an environment variable as I described in the previous section, which means you can use them to conditionally change request and response headers. As before, the idea is to check for a condition using ModSecurity and set an environment variable if the condition is met.

In the following example I use ModSecurity to instruct `mod_headers` to delete the session cookie:

```
# Simulate a condition that would want us
# to force the user to use another session
SecRule ARGS attackPattern \
    "phase:2,t:none,log,pass,setenv:DISABLE_OUTBOUND_SESSION"

# Expire session cookies when instructed
Header set Set-Cookie "PHPSESSID=;expires=Fri, 31-Dec-1999 00:00:00 GMT" \
    env=DISABLE_OUTBOUND_SESSION
```

Securing session cookies

In web applications that support user authentication, session cookies function as temporary passwords. Users provide their credentials only once and, assuming they are correct, their sessions are marked as authenticated. From that point on, whoever knows a session's ID can exercize full control over it. Great care needs to be taken when constructing session cookies to ensure that they are secure. In many applications, the security of session cookies can be improved by changing two aspects of how they are constructed:

Use of the httpOnly flag
: HttpOnly is an Internet Explorer innovation, which aims to prevent access to session cookies from JavaScript (which is the most common way to steal a session ID after a successful XSS attack). The idea is that session cookies are only needed by the server-side code, and that we lose nothing by forbidding access from JavaScript. With the httpOnly flag in place, session hijacking becomes significantly more difficult.

Use of the secure flag
: When a site uses SSL there is no way for an attacker to gain access to the data that is being exchanged between the site and the users. A frequent omission, when using SSL, is to omit to mark the session cookies as secure. The omission can lead to a compromise of users' session cookies, giving the attacker complete access to the corresponding sessions.

If you are using Apache 2.2.4 or better you can fix the above problems quickly, using just two `mod_headers` instructions. The following example improves the security of the session cookies used by PHP:

```
# Add missing httpOnly flag
Header edit Set-Cookie "(?i)^(PHPSESSID=.+(?!httponly))" "$1; httpOnly"

# Add missing secure flag
Header edit Set-Cookie "(?i)^(PHPSESSID=.+(?!secure))" "$1; secure"
```

The general idea is that we look at the Set-Cookie header, which is used to create new cookies, and look for the session cookies that do not have the desired flags set. If such incorrectly-set cookies are found, we modify the headers to append the flags. The example uses several rarely-used, but very useful features:

- The regular expression patterns both begin with (`?i`), which ensures that matching is case-insensitive.
- In the second part, a *negative lookahead assertion*, causing the entire pattern to match only if the bits in the assertion do not appear anywhere in the header.
- The fourth parameter, which contains the value that will replace an existing `Set-Cookie` value, makes use of backreferences (`$1`), which are replaced by the existing header value.

It's interesting how much information can be contained on a single line of text, isn't it!

Advanced Blocking

Simple blocking is straightforward.

Using status with deny and redirect.

Configuring a custom response page (separate section, or part of application setup?).

Not possible to block from phase 5.

Blocking from phases 3 and 4 may be problematic.

Plus there may be issues with mod_deflate.

Honeypot blocking (redirect per IP address/session).

Specify when to block but not how.

Delayed (phase) blocking — set a flag the continue to evaluate the rules in the phase. Maximise detection.

Score-based blocking (per transaction).

Persistent score-based blocking (per IP address, session, user).

External blocking.

What use is severity?

Can we use tags in blocking?

Reputation databases (rbl) for blocking.

GeoIP for blocking.

Making the most of regular expressions

Although ModSecurity supports many operators, regular expressions are so powerful and versatile that they remain the most-often seen choice in rules. ModSecurity uses the Perl Compatible Regular Expressions library [http://www.pcre.org], better known as PCRE. This is a well-known and widely-used regular expression library, and it is also used by Apache. Because they are so powerful, regular expressions will often surprise you, and you'll realise that they are more capable than you thought. This section will highlight the most important aspects of PCRE and the way this library is used in ModSecurity, but it is only the tip of the iceberg. I highly recommend that you familiarize yourself with the PCRE documentation [http://www.pcre.org/pcre.txt], which contains everything you need to know.

How ModSecurity Compiles Patterns

Regular expression patterns are compiled (converted into efficient internal representation) before they are used. The compilation step helps the library improve performance, doing as much work as possible only once, at configure-time. The compilation flags affect how patterns are used and you need to be aware about them. In the most important place where regular expressions are used, the `@rx` operator, ModSecurity uses two compilation flags:

`PCRE_DOLLAR_ENDONLY`
: Also by default, a dollar metacharacter will match a newline at the end of a string. Users often find this unexpected, and it messes up with the rules that want to have complete control over what is allowed in certain places. By using `PCRE_DOLLAR_ENDONLY` to compile patterns, the dollar character is made to match only at the end of the input.

`PCRE_DOTALL`
: By default, a dot metacharacter in a pattern matches all characters except those indicating newlines. In a security context that opens for a potential weakness where an attacker is able to use a newline to break up his attack payload and prevent a pattern

from matching. With PCRE_DOTALL set, a dot metacharacter will genuinely match any character.

Now that you know which compilation flags are used, it is important to learn about two that are *not* used:

PCRE_CASELESS
: Enabled case-insensitive matching. Because this flag is absent when the @rx patterns are compiled, all patterns are case-sensitive. (Use the t:lowercase transformation function to achieve case-insensitive matching, or read the next section, which shows another way.)

PCRE_MULTILINE
: This flag changes the behaviour of the ^ and $ metacharacter to force them to match at a beginning of a line and at an end of a line, respectively. Without it, PCRE will treat the entire input string as a single line. The PCRE default is used for the @rx operator. That means that a ^ metacharacter will *always* match at the beginning of the string, and $ will *always* match at the end.

There are several other places where regular expressions are used and, while they are not as security-sensitive as the @rx operator, you should still be aware of how they are compiled. Table Table 9.1, "Pattern compilation flags" gives a complete picture.

Table 9.1. Pattern compilation flags

Usage	Compilation flags used
@rx	PCRE_DOLLAR_ENDONLY, PCRE_DOTALL
@verifyCC	PCRE_DOTALL, PCRE_MULTILINE
SecAuditLogRelevantStatus	PCRE_DOTALL
SecRuleRemoveByMsg	No flags used
Variable selection (e.g. ARGS)	PCRE_CASELESS, PCRE_DOLLAR_ENDONLY, PCRE_DOTALL

Changing how patterns are compiled

If you are not happy with how ModSecurity compiles patterns, you'll be happy to hear that PCRE allows you to override the compile flags from within the pattern itself. For example, the following rule, which does not use any transformation functions, will match the word attack no matter of the case used:

```
SecRule ARGS "(?i)attack" phase:2,t:none
```

The (?i) part, placed at the beginning of the pattern, activates the PCRE_CASELESS flag for the entire pattern. It is also possible to change a setting only for a part of a pattern, by placing the modifier within.

```
SecRule ARGS "attack (?i)keyword" phase:2,t:none
```

The above will match attack keyword and attack KeYWORD, but not ATTACK keyword. If you place the modifier in a subpattern, the only the remainder of the subpattern will be modified:

```
SecRule ARGS "(key(?i)word) attack" phase:2,t:none
```

The above will match keyWORD attack, but not keyWORD ATTACK nor KeyWORD attack.

To remove a flag, use a dash in front of the letter. The following pattern unsets the PCRE_DOTALL flag that is used by ModSecurity by default:

```
SecRule ARGS "(?-s)keyword" phase:2,t:none
```

The complete list of the modifiers you can use in this way is in Table 9.2, "Pattern modifiers". For their complete meaning look them up in the PCRE documentation.

Table 9.2. Pattern modifiers

Modifier	Meaning
i	PCRE_CASELESS
J	PCRE_DUPNAMES
m	PCRE_MULTILINE
s	PCRE_DOTALL
U	PCRE_UNGREEDY
x	PCRE_EXTENDED
X	PCRE_EXTRA

Common pattern problems

Mistakes in regular expression patterns are common, but two are seen more often than others:

Forgetting to escape the dot metacharacter
: It is easy to forget that dot is a metacharacter that needs to be escaped. That most commonly happens when you're writing patterns to match IP addresses, which have many dots in them. An unescaped dot will match any character, matching against unintended characters if it was not meant to be used as a metacharacter.

Not using the ^ and $ anchors when matching entire input

The use of ^ and $ anchors is required when you want to match entire input. It ensure that whatever pattern you write, it matches everything. If you omit to use one or the other, you allow the attacker to send anything before your pattern (when you don't have a ^) and anything after your pattern (when you don't have a $). Without the anchors, a pattern may match a substring in the middle, ignoring anything else.

Regular Expression Denial of Service

Regular Expression Denial of Service (or ReDoS) is a relatively obscure problem that affects every regular expression writer. Some regular expression constructs are known to suffer from very bad (exponential) performance when certain edge cases are encountered. If you are not careful you can write a pattern that can be manipulated, from the outside, by an attacker, to consume most of all of your server's resources.

Here are some examples of vulnerable patterns borrowed from Alex's and Adar's presentation (see below):

- `(a+)+`
- `([a-zA-Z]+)*`
- `(a|aa)+`
- `(a|a?)+`
- `(.*a){x}`, for x > 10

For more information on this subject look up the following presentations:

- *Regular Expression Denial of Service* (2003), by Scott A. Crosby.
- *Regular Expression Denial of Service* (2009), by Alex Roichman and Adar Weidman.

Resources

Don't be surprised if you sometimes get overwhelmed working with regular expressions. That's entirely normal and will go away in time. You don't have to buy a book in order to become proficient in regular expressions, but it will certainly help if you do. My only issue with the available books is that they all cover many regular expression flavours, and I am only interested in PCRE. There are at least two books you should look at:

- *Mastering Regular Expressions*, by Jeffrey Friedl (O'Reilly, 2006), is widely considered to be a classic work on regular expressions.

- *Regular Expressions Cookbook*, by Jan Goyvaerts and Steven Levithan (O'Reilly, 2009) is a recent addition to the regular expression work, and adopts a more practical style of learning.

You should also look at one of the available tools, as they will enable you to interactively design and analyse regular expressions:

- RegexBuddy [http://www.regexbuddy.com], a commercial tool written by the Regular Expression Cookbook co-author Jan Goyvaerts, is often recommended as the ultimate regular expression assistant.
- The Regex Coach [http://weitz.de/regex-coach/], a free tool written by Edi Weitz.
- Expresso [http://www.ultrapico.com/Expresso.htm] is a free tool (requires registration) from Ultrapico.

Performance tips

Explain performance: which operations are expensive. The overhead of the rule engine.

Single rule vs multiple rules.

Regular expressions vs parallel matching.

Combining regular expression patterns.

Fail-fast techniques: evaluate a condition than skip over a bunch of rules, evaluate a condition than dig deeper.

Whitelist requests.

Restrict full transaction logging (audit logging). How much does audit logging cost?

No debugging in production; set to zero.

Minimal rule set

Turn off inspection where not necessary (especially response body inspection)

Monitor memory requirements

Be reasonable in what you expect

Some benchmarks?

How does ModSecurity affects web server? What about the Core Rules?

Performance testing using special compilation mode.

Keeping track of performance in the logs.

10 Content Injection

Content injection is an innovative security technique that allows you to inject arbitrary content into HTTP response bodies. The technique was designed to address the threats that take place in the browser itself, and thus extend the reach of a web application firewall from the server-side. With content injection, a server can reach out to inject dynamic content (JavaScript) into responses, gaining in-browser inspection capabilities. The idea is that you first perform your normal server-side inspection, then you inject JavaScript into the HTTP response to continue the inspection there, with full access to the browser's internal state. This section will give you a good overview of several useful and easy to use techniques based on content injection.

> **Note**
>
> There is nothing to say that content injection has to be used for defense. There is a school of thought that says that offense is the best defense. If you subscribe to that view, you could use content injection to attack the attackers back, injecting malware directly into their browsers. Just make sure you understand your legal position before you do anything that might be crossing the line.

Writing Content Injection Rules

Content injection allows you to inject content, possibly on per-response basis, either at the beginning of a response, or at the end. Injecting at the beginning is useful if you want to attempt to prevent attacks. Injecting at the end is useful if you want to inspect the content of the page and the state after all other JavaScript code has already been run.

To start you will need to enable the injection feature using the `SecContentInjection` directive:

```
# Enable content injection
SecContentInjection On
```

> **Note**
>
> Content injection does not require that you have `SecResponseBodyAccess` enabled. It works equally well with this setting enabled or disabled.

In the next step you need to determine whether injection would make sense. Web servers process many types of requests, and only a fraction of the corresponding responses can be injected. You wouldn't, for example, want to inject anything into an image, because it would be corrupted. You should check the response content type in phase 3 (use `RESPONSE_CONTENT_TYPE`, which has direct access to Apache's internal variables, rather than `RESPONSE_HEADERS:Content-Type`, which may not be always available) to determine if it uses one of the modifiable responses. You can use the following framework for all your content injection rules:

```
SecContentInjection On

# First check if we should inject anything
SecRule RESPONSE_CONTENT_TYPE !^text/html \
    phase:3,nolog,pass,skipAfter:HTML_CONTENT_INJECTION_DONE

# ... your content injection rules here

SecMarker HTML_CONTENT_INJECTION_DONE
```

First you check for the correct content type, jumping over all your content injection rules if an incorrect type is used in a response. If you are going to inject into more than one type of document (e.g., `text/plain` and `text/html`) then you are probably going to need to use different rules, with different content for each type. In that case, just repeat the above fragment, making sure to choose the content type correctly and to use a unique `SecMarker` value in each group.

Finally, to inject content, use the prepend and append actions. The following example injects a header and a footer into a HTML response:

```
SecAction phase:3,nolog,pass,prepend:'Header<hr>'
SecAction phase:3,nolog,pass,append:'Footer<hr>'
```

> **Note**
>
> The content injection facilities will not perform any output encoding, which means that you must manually encode everything that you want injected. The `append` and `prepend` actions do support variable expansion, enabling you to inject dynamically generated content, but you *must* take care to *never* inject any user-controlled content. Doing so would create a XSS vulnerability! Only inject what you have 100% control over.

For testing purposes you can also try this simple JavaScript code, which will write the URL of the current page:

```
SecAction phase:3,nolog,pass,prepend:…
'<script>document.write(document.location)</script>'
```

So we've established that you can have your JavaScript code inside the browser. But what can you do with this ability? Here are some ideas.

- Inspect request parameters, including the fragment identifier, which is normally not sent to servers.
- Inspect browser state. For example, a popular technique used to assist in XSS attacks is to store payload in window names (property `window.name`). That field is out of bounds to a server, but not to the injected JavaScript code.
- Inspect browser configuration, for example look for vulnerable plug-ins.
- Inspect page state and structure (DOM) at the end of page execution.
- Redefine the built-in JavaScript functions to detect unusual activity patterns.

JavaScript is a fascinating language that is endlessly tweakable. Describing advanced JavaScript attacks is out of the scope of this book. If you want to go there, simply pick up the most advanced JavaScript book you can find and use it as a starting point.

CRSF Defense Using Content Injection

The most advanced and imaginative use of the content injection feature is that devised by Ryan C. Barnett, the ModSecurity Community Manager and author of the Core Rule Set. He established a way to use content injection to defend vulnerable applications against Cross-Site Request Forgery (CSRF) attacks, otherwise only possible through the modification of the source code of the vulnerable applications. (If you are not familiar with CSRF, I suggest that you read through the CSRF entry on Wikipedia [http://en.wikipedia.org/wiki/Cross-site_request_forgery].)

The usual way to defend against CSRF is to embed special tokens into application forms, and accept only those submits that contain the correct token values. CSRF requests faced with such defenses always fail, because they have no way to "know" the correct token value.

Ryan's approach was to use content injection to inject JavaScript into all application pages, which is then used to modify all page forms to add tokens where they wouldn't normally exist. In the second part of the trick, he would have ModSecurity rules inspect all `POST` requests to verify that they contain the correct values. Brilliant!

For more information, look up Ryan's Black Hat DC 2009 whitepaper *WAF Virtual Patching Challenge: Securing WebGoat with ModSecurity*. The 26-page document contains many other interesting techniques.

Communicating back to the server

When you detect a problem using JavaScript you need to somehow communicate the fact back to the server. The best way to do that is to get the browser to send a special request. The simplest way to do that is by writing some HTML into the response:

```
document.write("<script src=/security-error.js><" + "/script>");
```

> **Note**
>
> The injected payload must never contain the characters `</script>` anywhere except at the very end. If it does, then that's where browsers will terminate the entire payload, and probably even cause a JavaScript error. String concatenation, as used in the above example, is often used to deactivate the closing tag.

To the special request you should add a rule that detects it and raises an alert in ModSecurity. While you're there, you may consider doing other things, for example, cancelling the victim's application session. You may want to consider including an error code in the request (e.g., as a parameter), which will help you in establishing exactly where the problem was. You should also be prepared that this communication mechanism can be discovered and subverted by the attacker. In that light, don't use any information obtained from such requests for anything apart from logging.

Interrupting page rendering

Perhaps you'll think that detection is not enough and that you will need to prevent in-browser attacks. JavaScript does not offer a way to stop page rendering, but you can do the next best thing—redirect the user someplace else using `location.replace()`. In my tests, the invocation has the effect of a effectively stopping rendering and moving elsewhere. For example:

```
location.replace("http://www.example.com/security-error.html");
```

The stopping of page rendering when `location.replace()` is invoked is a side-effect and you should generally not expect it to work across all browsers, or that it will continue to work in the browsers it works in today. For example, some browser may continue to process JavaScript while another page is being loaded. You should allow for a fact that some attacks may get through.

If you choose to implement prevention in this way, don't forget to put some user-friendly explanation for the sudden redirection to the page with an unusual name. Your users will appreciate it. The advantage of using prevention like this is that it also notifies you of the

problem. Whenever someone access that special page you will know that they have been attacked.

Using external JavaScript code

In the current implementation of content injection, you are limited to the content you can put in a parameter to the append and prepend actions. In particular, you won't be able to inject any non-printable characters. You can escape a single quote with a backslash, but that's the only escape option ModSecurity supports at present. If you do run into trouble you can always store the JavaScript code in a separate file and only inject a link to it.

If you can place a file onto the root of the web site that is being protected, use this:

```
SecAction "phase:3,nolog,pass,prepend:'<script src=/ids.js></script>'"
```

If you have several web sites and you'd like to use one file for all of them, use an absolute address:

```
SecAction "phase:3,nolog,pass,prepend:…
'<script src=https://www.example.com/ids.js></script>'"
```

Finally, if you need to construct the address dynamically, you can do that by injecting JavaScript that will generate the HTML code that is needed to include the external JavaScript:

```
SecAction "phase:3,nolog,pass,prepend:…
'<script>document.write(unescape(\"<script src=\'\" + document.location.protocol …
+ \"//www.example.com/ids.js\'%3c/script>\"));</script>'"
```

Several aspects of the above rule need explaining:

1. You use `document.write()` to output HTML to the document body.
2. Make sure to escape all single and double quotes in the code.
3. In the above example I used URL encoding (converting the opening angle bracket to `%3c`) in combination with `unescape()` to deactivate the closing script tag. This approach can also used if you need special characters in JavaScript (and you cannot write them directly because of ModSecurity's poor escape syntax).
4. The code uses the `document.location.protocol` property, which will be `http:` for plain-text connections and `https:` for the encrypted ones, to construct an URL that will correspond to the security level of the including document. That will help with the performance on non-SSL sites.

Communicating with Users

Another interesting application of content injection is talking to the application users. Ages ago I wrote some code, practically as a party trick, that would detect access using vulnerability scanners (e.g., Paros Proxy) and send a message back that we don't like being probed.

Such a rule can be as simple as this example:

```
SecRule REQUEST_HEADERS:User-Agent Paros \
    "phase:3,pass,prepend:'Use of Paros Proxy is strictly forbidden'"
```

If your site uses sessions and you've taught ModSecurity to track them, you can send per-session messages that expire after a period of time. I will show you how to do that, using an example that detects the word `attack` anywhere in request parameters (let's pretend that we are detecting an SQL injection attack), then sets a message that will be displayed to the same session for 60 seconds (even in the requests that do not contain the attack). The example consists of only two rules.

The first rule is used to trigger the message:

```
# The following rule triggers a message. Session must have been
# established (using setsid) beforehand, otherwise the execution
# of this rule will cause an error.
SecRule ARGS attack "nolog,pass,msg:'SQL Injection is lame',…
setvar:SESSION.message_flag=1,expirevar:SESSION.message_flag=60,…
setvar:SESSION.message=%{RULE.msg}"
```

The detection itself is trivial, but the rest needs an explanation:

1. The first `setvar` action (`setvar:SESSION.message_flag=1`) creates a per-session flag that is used to indicate that a message exists.
2. The `expirevar` action (`expirevar:SESSION.message_flag=60`) is used to delete the `SESSION.message_flag` variable after 60 seconds.
3. The second `setvar` action (`setvar:SESSION.message=%{RULE.msg}`) is used to define the message itself. Here I used a little trick, using variable expansion to use the rule message (as defined with `msg:'SQL Injection is lame'`) as the session message. I did this because I needed my message to contains spaces, yet the syntax for variable setting does not allow it (rule messages can be enclosed in single quotes, and can thus contain whitespace). The `RULE` collection is a special collection that always refers to the current rule.

The second rule is used to detect the presence of `SESSION.message_flag`, and display the message stored in `SESSION.message`:

```
# The following rule displays the message. As before, the prepend action
```

```
# must be executed only if the response content type is right.
SecRule SESSION.message_flag "@eq 1" phase:3,nolog,pass,prepend:%{SESSION.message}
```

When, after 60 seconds, the `expirevar` statement from the first rule kicks in, the `SESSION.message_flag` variable will be deleted and the message will go away.

11 Writing Rules in Lua

The ModSecurity Rule Language is relatively easy to use, but it is fairly limited. After all, the directives have to obey the Apache configuration syntax, so there is only so much we can do within those boundaries. I like to think that you can use the rule language to get 80% of your tasks done, and quickly too: common things are simple to do, complex things are possible. At some point, however, the rule language stops being an appropriate tool for the task, and you need to look elsewhere. Starting with ModSecurity 2.5 you can write rules in Lua [http://www.lua.org], a fast and memory-efficient scripting language. These attributes made it very popular with game programmers, who are always trying to get that extra ounce of performance.

The advantage of Lua is that it is a proper programmer language, which means that you are only limited by your programming skills. The disadvantage, as you might expect, is a performance penalty. Some of that penalty comes from the fact that Lua scripts need to be interpreted at run time, and some because the current implementation in ModSecurity (version 2.5.x) is not as efficient as it can be. Having said that, I think the performance is going to be adequate provided you only run a small number of Lua rules per transaction.

> **Note**
>
> The support for Lua in ModSecurity is experimental. The code itself is stable and production-ready, but the APIs may change. A future version of ModSecurity may make changes to the way things are implemented and that you may need to update your rules to make them compatible. Other than that, there is no reason not to use Lua rules.

There are two ways in which Lua can be used to enhance your rule sets. First, you can write detection rules in it. Second, you can write scripts that are executed on a rule match. The remainder of this section explains both of these features.

Rule Language Integration

Although in the previous section I made it sound like Lua rules are separate from the rule language, that's not actually true. In ModSecurity Lua is implemented as an rule language extension, via the `SecRuleScript` directive. For example, this is how you run a Lua script:

```
SecRuleScript /path/to/script.lua phase:2,log,deny
```

Comparing to the SecRule directive, the variables and the operators are gone. They are replaced with a single parameter, which is the location of the Lua script you wish to run. That means that the script will choose which variables it wishes to inspect and in which order. The action list is still there, though. You can see that the rule in the above example runs in phase 1, and that it logs and blocks on a match. It will probably not surprise you when I tell you that the one task of a Lua rule is the same as any other rule—to inspect one or more variables, in any way it chooses, and deliver a pass/fail verdict.

Lua Rules Skeleton

Every Lua rule needs to have an entry point that ModSecurity can find—the main function. This is what the simplest Lua rule looks like:

```
function main()
    -- Never match
    return nil;
end
```

As you suspect, the above rule does not do much. It only returns `nil`, which means that there is no match. For a Lua rule to match, it needs to return a message:

```
function main()
    -- Always match
    return "Error message";
end
```

The beauty of the way Lua is integrated with ModSecurity is that, once you return an error message, the rule language takes over and processes the action list. Thus, with Lua rules you still get to use what you already know. For example, all the metadata information you provide for Lua rules in the exactly same way as you do for normal rules:

```
SecRuleScript /path/to/script.lua \
    phase:2,log,deny,id:1001,rev:1,severity:3
```

Whatever you can do with a SecRule directive you can do with `SecRuleScript`. There's no limit to what is possible, including using Lua in rule chains.

Accessing Variables

Once inside a Lua rule, the first thing you will need to do is access some variables. The following example retrieves two variables from ModSecurity:

```
function main()
    -- Retrieve remote IP address
    local remote_addr = m.getvar("REMOTE_ADDR");

    -- Retrieve username
    local username = m.getvar("ARGS.username", {"lowercase"});

    if ((username == "admin") and (remote_addr ~= "192.168.1.1")) then
        return "Admin sign-in not allowed from IP address: " .. remote_addr;
    end

    -- No match
    return nil;
end
```

A call to the `m.getvar()` function will retrieve the variable named in the first parameter. In the example, the value of `REMOTE_ADDR` is retrieved and placed into the Lua variable `remote_addr`.

The function has an optional second parameter. If used, it must contain a list of transformation functions that will be applied to the variable before it is returned to Lua. In the example, the value of `ARGS.username` is retrieved from ModSecurity, passed through the lowercase transformation function and placed into the Lua variable `username`.

It is also possible to retrieve more than one variable at once, but for that you use the `m.getvars()` function (note the additional `s` in the name). The following example retrieves all request parameters, then examines them one at a time.

```
function main()
    -- Retrieve all parameters
    local vars = m.getvars("ARGS", {"lowercase", "htmlEntityDecode"});

    -- Examine all variables
    for i = 1, #d do
        -- Examine one value
        if (string.find(d[i].value, "<script")) then
            return ("Suspected XSS in variable: " .. d[i].name .. ".");
        end
    end

    -- Nothing wrong found.
    return nil;
```

```
end
```

The `m.getvars()` function works differently. It does not return just the value of the requested variable. Instead, it returns an object with two members: `name`, which contains the name of the variable, and `value`, which contains the corresponding value. The above example demonstrates how both are used.

Logging

Sometimes a Lua rule will not be working as you expect, but you won't have any clues as to why. You can troubleshoot your scripts by emitting debug log messages, using the `m.log()` function.

```
function main()
    -- Log something
    m.log(4, "Hello World from Lua!");

    -- Never match
    return nil;
end
```

The `m.log()` function takes two parameters, first of which is the desired log level and the second the desired message.

Lua Actions

With the addition of Lua, the exec action was extended to support Lua natively. Normally, you supply every instance of the exec action with a path to an external script and ModSecurity executes that script in a separate process whenever it needs to. If the script path ends with .lua, however, ModSecurity will process the script using the embedded Lua interpreter. This approach not only achieves better performance (no need to start a new process), but also gives the Lua script access to the current transaction context.

```
SecRule ARGS test phase:2,log,pass,exec:/path/to/script.lua
```

There's only one rule for the script, and that is to define the same entry point as all other Lua scripts. There is no need to return anything from the `main()` function.

```
function main()
    -- Log something
    m.log(4, "Lua executed in exec!");
end
```

Now, the example looks deceptively simple, so much that you may wonder of what use could Lua possibly be. The answer is that you can do from Lua pretty much anything you want. You not only get the programming language and the standard Lua libraries, but you also get access to a number of extensions that take care of filesystem access, sockets, database access, and so on. And, because Lua scripts executed in this way have access to the transaction context and the persistent storage, what you have is a seamless scripting extension of ModSecurity.

12 Handling XML

ModSecurity has very good XML support, which is made possible by a tight integration with LibXML2 [http://xmlsoft.org]. LibXML2 is one of the fastest XML libraries available, making it very suitable for the performance-sensitive work in ModSecurity. The integration is seamless, effectively making XML payloads just another source of data, to which you can apply your usual rule-writing techniques. The following functionality is supported:

- XML parsing
- DTD validation
- XML Schema validation
- XPath expressions

Once upon a time it was possible to leave out XML functionality when compiling ModSecurity, but newer versions do not support that any more. You should reasonably expect for the XML processing features to be available in ModSecurity.

> **Note**
>
> You don't want to use ModSecurity as an XML testing tool, because the entire cycle (write rules, then send payload, then analyze debug log) is very slow. You should instead use an XML validation tool. Probably the best option is `xmllint`, because it is based on the same library used by ModSecurity.

The examples used in this section were adapted from the sample written by Steve Traut for the XMLBeans project [http://xmlbeans.apache.org].

XML Parsing

Although ModSecurity is capable of parsing XML, it won't attempt any parsing by default. XML parsing is very resource consuming and many installations do not need it. Even when they do, recognizing that XML parsing is needed is not something that can be easily done

in a portable way. Other request body processors (URLENCODED and MULTIPART) rely on using a standardized content type for detection when they are needed, but there is no such thing for XML.

To enable XML parsing you'll have to go through the manual request body processor activation. There are two things you will need to do:

1. Analyze request to determine if XML parsing is needed. Most requests won't need XML parsing enabled. Figuring out which do will depend on the exact content type used by your application. In many cases the Content-Type header will contain text/xml, and that is what I will assume in my examples.
2. Instruct ModSecurity to use the XML request body processor for the requests that do need it

For example:

```
# Detect XML payloads and activate XML parsing
SecRule REQUEST_HEADERS:Content-Type ^text/xml$ \
    phase:1,t:none,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML
```

The example uses the @rx operator, but a straightforward @streq would have worked too. Notice how I used t:lowercase to ensure the comparison is case insensitive (which is always appropriate when working with Content-Type).

It is important to always use phase 1 (REQUEST_HEADERS) when determining request body processors. Request body parsing is done right after phase 1 completes and the processor choice must be made before then.

When you're writing ModSecurity rules you usually have to test a lot, and when you're working with XML you will have to test even more. For my testing I use the little utility script that was distributed with ModSecurity 1.x, called run-test.pl. While this script isn't distributed with ModSecurity 2.x., you can still get it directly from the repository:

```
http://mod-security.svn.sourceforge.net/viewvc/mod-security/m1/trunk/util/…
run-test.pl?revision=6
```

With this script in hand, you can construct and send raw HTTP requests to your web server to test your rules. For example, I used the following file (which I named xml.t) to test XML parsing:

```
POST / HTTP/1.0
Content-Type: text/xml
Content-Length: 633

<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>
```

To send a file to a web server, you specify the server information (in the example below, both the IP address and the port) and the file you wish to send. I also often use the debug switch (`-d`), which makes the tool output all traffic to standard output:

```
./run-test 192.168.3.100:8080 -d xml.t
```

Let's see how ModSecurity processed this test request. Below is the debug log output at level 8 (which is shorter than the level 9 output, but equally meaningful in this case).

First, the rule ran in phase 1 to check the value of the `Content-Type` request header. It matched, causing the `ctl` action to set the request body processor to `XML`.

```
[4] Recipe: Invoking rule 8de36c8; [file "/home/ivanr/apache/conf/httpd.conf"] …
[line "518"].
[5] Rule 8de36c8: SecRule "REQUEST_HEADERS:Content-Type" "@rx ^text/xml$" …
"phase:1,auditlog,t:none,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML"
[4] Transformation completed in 24 usec.
[4] Executing operator "rx" with param "^text/xml$" against …
REQUEST_HEADERS:Content-Type.
[4] Operator completed in 54 usec.
[4] Ctl: Set requestBodyProcessor to XML.
[4] Warning. Pattern match "^text/xml$" at REQUEST_HEADERS:Content-Type. …
[file "/home/ivanr/apache/conf/httpd.conf"] [line "518"]
[4] Rule returned 1.
```

Then we see the second phase starting and ModSecurity reading the request body and forwarding it to the XML parser.

```
[4] Second phase starting (dcfg 8dfee38).
```

```
[4] Input filter: Reading request body.
[4] XML: Initialising parser.
[4] XML: Parsing complete (well_formed 1).
```

The last line indicates the completion of XML parsing. It also indicates that the XML was well formed. It it weren't, there message would display a `0` instead of the `1`. This message makes a good point, actually: You not only need to enable XML parsing, but also verify that it was successful.

To verify how XML parsing went in a rule, you use the `REQBODY_PROCESSOR_ERROR` variable, as you do with all request body processors. I covered this topic in detail in the section called "Handling Parsing Errors". If you follow my advice from that section and use the rule to check for request body processors errors (also reproduced below), you will have already been covered for XML parsing errors too:

```
# Verify that we've correctly processed the request body.
# As a rule of thumb, when failing to process a request body
# you should reject the request (when deployed in blocking mode)
# or log a high-severity alert (when deployed in detection-only mode).
SecRule REQBODY_PROCESSOR_ERROR "!@eq 0" \
    "phase:2,t:none,log,block,msg:'Failed to parse request body: …
%{REQBODY_PROCESSOR_ERROR_MSG}'"
```

We can easily check if that is correct. Make a copy of `xml.t`, calling the new file `xml-invalid.t`, then replace one of the angle brackets with a space. (Replacing a character will ensure the payload length remains the same. If you add or remove a character you will need to update the `Content-Length` request header to reflect the change.) When you send such modified file to the server, the debug log will report the problem:

```
[4] Second phase starting (dcfg 8df6e68).
[4] Input filter: Reading request body.
[4] XML: Initialising parser.
[4] XML: Parsing complete (well_formed 0).
[2] XML parser error: XML: Failed parsing document.
```

Then, a few lines down in the log file, you will see the second rule triggering:

```
[4] Recipe: Invoking rule 8e5e538; [file "/home/ivanr/apache/conf/httpd.conf"] …
[line "522"].
[5] Rule 8e5e538: SecRule "REQBODY_PROCESSOR_ERROR" "!@eq 0" …
"phase:2,status:500,t:none,log,block,msg:'Failed to parse request body: …
%{REQBODY_PROCESSOR_ERROR_MSG}'"
[4] Transformation completed in 3 usec.
[4] Executing operator "!eq" with param "0" against …
REQBODY_PROCESSOR_ERROR.
[4] Operator completed in 8 usec.
[4] Rule returned 1.
```

```
[1] Access denied with code 500 (phase 2). Match of "eq 0" against …
"REQBODY_PROCESSOR_ERROR" required.  [file "/home/ivanr/apache/conf/httpd.conf"] …
[line "522"] [msg "Failed to parse request body: XML parser error: XML: Failed …
parsing document."]
```

> **Note**
>
> Just because an XML payload is not well-formed, does not mean that your subsequent rules are not going to run. They will run, but they will only have access to a partial XML tree, created until the parsing error was encountered. What this tree will contain depends on the nature of the error. If you choose not to block on a request body processor failure, then you need to at least ensure that you don't rely on the results of your subsequent XML rules. For example, you could evaluate `REQBODY_PROCESSOR_ERROR` again and skip over them. If you don't mind working with a partial XML payload, or even if that's desired, then you don't need to do anything.

DTD Validation

Sometimes you will be happy to work with a partial (invalid) XML payload, but some other times you will want to perform further validation. The validation requires one further rule, in which you specify the type of validation and the file that contains the rules:

```
SecRule XML "@validateDTD /path/to/apache2/conf/xml.dtd \
    "phase:2,log,block,msg:'Failed to validate XML payload against DTD'"
```

The file `xml.dtd`, which contains a DTD for the XML payload used earlier in this section, contains the following:

```
<!ELEMENT phone (#PCDATA)>
<!ATTLIST phone location CDATA #REQUIRED>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT address (street, city, state, zip)>
<!ATTLIST address location CDATA #REQUIRED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT employee (name, address+, phone+)>
<!ELEMENT employees (employee)>
```

When you submit the same XML payload as before, you get:

```
[4] Recipe: Invoking rule 8265d30; [file "/home/ivanr/apache/conf/httpd.conf"] …
[line "525"].
[5] Rule 8265d30: SecRule "XML" "@validateDTD /home/ivanr/apache/conf/xml.dtd" …
```

```
"phase:2,status:500,log,block,msg:'Failed to validate …
XML payload against DTD'"
[4] Transformation completed in 1 usec.
[4] Executing operator "validateDTD" with param "/home/ivanr/apache/conf/xml.dtd" …
against XML.
[9] Target value: "[XML document tree]"
[4] XML: Successfully validated payload against DTD: /home/ivanr/apache/conf/xml.dtd
[4] Operator completed in 612 usec.
[4] Rule returned 0.
```

> **Note**
>
> The `@validateDTD` operator returns a match if it fails to validate, and no match if everything is all right.

When validation fails, the error messages from LibXML2 will be recorded as notices (level 3), which means that they will appear in the debug log, audit log and in the Apache error log. For example, when I changed the payload to transmit employee name with the first name and last name separately:

```
<firstname>Fred</firstname>
<lastname>Jones</lastname>
```

I got three LibXML2 errors in return:

```
[3] Element employee content does not follow the DTD, expecting (name , …
address+ ,  phone+), got (firstname lastname address address phone phone phone )
[3] No declaration for element firstname
[3] No declaration for element lastname
```

And there was also one fatal error from the validation rule itself:

```
[1] Access denied with code 500 (phase 2). XML: DTD validation failed. [file …
"/home/ivanr/apache/conf/httpd.conf"] [line "525"] [msg "Failed to …
validate XML payload against DTD"]
```

> **Note**
>
> You must ensure that you enter the correct path to the DTD which you need for validation. If the path is incorrect validation will fail silently, without a match. This is a flaw in ModSecurity, which is planned to be fixed in v2.6 (ticket MODSEC-12 in the Tracker).

XML Schema Validation

The XML Schema validation rule is functionally identical to that used for DTD validation:

```
SecRule XML "@validateSchema /path/to/apache2/conf/xml.xsd \
    "phase:2,log,block,msg:'Failed to validate XML payload against schema'"
```

XML Schemas allow for much stricter validation, but the rule files are much more complicated. Below is the XML Schema equivalent of the DTD used in the previous section:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://xmlbeans.apache.org/samples/xquery/employees"
  xmlns="http://xmlbeans.apache.org/samples/xquery/employees">
  <xs:element name="employees">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="employee" type="employeeType" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="employeeType">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="address" type="addressType" maxOccurs="unbounded"/>
      <xs:element name="phone" type="phoneType" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="addressType">
    <xs:sequence>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:NCName"/>
      <xs:element name="state" type="xs:NCName"/>
      <xs:element name="zip" type="xs:integer"/>
    </xs:sequence>
    <xs:attribute name="location" type="xs:NCName" use="required"/>
  </xs:complexType>
  <xs:complexType name="phoneType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="location" type="xs:NCName" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

LibXML2, the underlying XML library used by ModSecurity, is known not to fully implement the XML Schema standards. You may encounter validation problems that are not a result of a problem in a request, but a result of the incomplete XML Schema implementation in LibXML2. In that case, your best bet is to try to upgrade the library to a newer version

(ModSecurity will use the same library version as used by your operating system). If that does not help, try seeking help on the LibXML2 users mailing list.

XML Namespaces

Initially, XML was simple and easy to understand, like the one example I've many times used in this section. As it gained in popularity, however, people decided that they wanted to combine XML documents of different types and needed a way to distinguish which elements belong to which types. XML namespaces were born. You've already seen a namespace in the one XML Schema we used so far, but that document only used one namespace.

To demonstrate how namespaces work, I have reworked the original example to split it into two namespaces. One for the `employees` element and the other for the `address` element:

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

In order to use a namespace you choose a prefix (it can be anything) you associate it with a namespace URI. In the above example the prefix is `a` (nice and short) and the URI is `http://www.example.org/address` (it's not necessary for the URI to work, however, it's just a unique identifier). Once a namespace has been introduced, you need to rewrite all the tags that belong to it to use the prefix.

Of course, the original XML Schema we used for validation won't work any more. The assumption, with the new XML payload, is that two schemes are needed. The address schema (`xml-address.xsd`) only defines the rules for addresses:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.example.org/address"
  xmlns="http://www.example.org/address">

  <xs:element name="address" type="addressType"/>

  <xs:complexType name="addressType">
    <xs:sequence>
      <xs:element name="street" type="xs:string"/>
      <xs:element name="city" type="xs:NCName"/>
      <xs:element name="state" type="xs:NCName"/>
      <xs:element name="zip" type="xs:integer"/>
    </xs:sequence>
    <xs:attribute name="location" type="xs:NCName" use="required"/>
  </xs:complexType>

  <xs:complexType name="phoneType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="location" type="xs:NCName" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>

</xs:schema>
```

The employees schema (`xml-employees.xsd`), defines the rules for everything else:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://xmlbeans.apache.org/samples/xquery/employees"
  xmlns="http://xmlbeans.apache.org/samples/xquery/employees"
  xmlns:a="http://www.example.org/address">

  <xs:import namespace="http://www.example.org/address" …
schemaLocation="xml-address.xsd"/>

  <xs:element name="employees">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="employee" type="employeeType" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="employeeType">
```

```
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element ref="a:address" maxOccurs="unbounded"/>
      <xs:element name="phone" type="phoneType" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="phoneType">
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="location" type="xs:NCName" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:schema>
```

Notice how this second schema uses the XML Schema import facility to refer to `xml-address.xsd`, and then uses the address element by reference.

When you need to validate a document that uses multiple schemas, like in the example above, the parameter you supply to `@validateSchema` must be the path to the main schema. You should also place all dependent schemas in the same directory as the main one.

XPath Expressions

XPath expressions are for XML what SQL is for relational database: a language that is used to extract the query data. They are very powerful and generally very easy to use. I've placed several examples in Table 12.1, "XPath expression examples", but if you have never worked with XPath expressions before I recommend that you go through this very nice tutorial on zvon.org [http://www.zvon.org/xxl/XPathTutorial/].

Table 12.1. XPath expression examples

XPath expression	Description
`/`	Root element
`/employees/employee`	All employees
`//address`	An address, under any parent element.
`//*`	All elements in payload
`/employees/employee/address[2]`	The second employee address
`//phone[@location='work']`	All work phone numbers

XPath expressions can only be used against the XML collection, and only in phase 2 (`REQUEST_BODY`). For example:

```
SecRule XML:/employees/employee/name/text() !^[a-zA-Z ]{3,33}$ \
    "phase:2,deny,msg:'Invalid employee name'"
```

Unless you've worked with XPath expressions in ModSecurity before, the results may not always be what you expect. Some XPath expressions will give you tidy results. For example, the one used in the previous example will return `Fred Jones`. But that only happens when you select a simple element, i.e. one that does not have any child elements. If the element you select has child elements, you get back everything it contains except the markup.

Try this, for example:

```
# Get the complete second employee address
SecRule XML:/employees/employee/address[2] TEST \
    "phase:2,deny"
```

The address fragment in the XML payload contains the following text (notice the whitespace, which I left the same as in the original payload):

```
        <address location="work">
            <street>2011 152nd Avenue NE</street>
            <city>Redmond</city>
            <state>WA</state>
            <zip>98052</zip>
        </address>
```

The debug log reveals what was used for matching:

```
[4] Recipe: Invoking rule 978d260; [file "/home/ivanr/apache/conf/httpd.conf"] …
[line "565"].
[5] Rule 978d260: SecRule "XML:/employees/employee/address[2]" "@rx TEST" …
"phase:2,log,auditlog,deny"
[4] Expanded "XML:/employees/employee/address[2]" to "XML|XML".
[4] Transformation completed in 2 usec.
[4] Executing operator "rx" with param "TEST" against …
XML:/employees/employee/address[2].
[9] Target value: "\n            2011 152nd Avenue NE\n            Redmond\n …
          WA\n            98052\n        "
```

You can see that the whitespace is all there, including the newline characters.

As a rule of thumb, when working with XML you should restrict yourself to the analysis of specific fields. Bulk-analysis (for example, using `//*`, which returns all elements in an XML payload) is just not going to very effective, because even smaller payloads will be broken into dozens and larger into possibly hundreds and thousands of small pieces. The performance of bulk XML matching is likely to be very bad. When the `//*` expression is used with our short XML example, it creates 16 variables.

XPath and Namespaces

Once you move away from simple XML documents to those using namespaces, your XPath expressions might stop to work. If the part of the XML uses prefixes, your XPath expressions won't match it any more. For example, we could have used this "clean" XPath expression to validate ZIP codes in the first XML example:

```
SecRule XML://address/zip !^\d+$ \
    "phase:2,deny,msg:'Invalid ZIP code'"
```

To get the rule working with an XML document that uses prefixes, like the second XML example, you could try to modify the XPath the expression to include the prefixes, but that will only cause XPath evaluation to fail because LibXML2 will try to match the prefix to a namespace, but it won't know how. You will get `XML: Unable to evaluate xpath expression` in the debug log. Even if LibXML2 didn't complain this approach wouldn't work because the choice of prefix is in the hands of the request sender. You don't get to control it on the server.

The solution is to use prefixes in XPath expressions, but also tell LibXML2 about the namespace, using the `xmlns` action:

```
SecRule XML://a:address/a:zip !^\d+$ \
    "phase:2,deny,msg:'Invalid ZIP code',xmlns:a=http://www.example.org/address"
```

The above will work as it would in the original example, returning two ZIP codes. It will even work if the sender chooses an entirely different prefix.

XML Inspection Framework

The validation examples so far all assumed one validation per request, but there was no framework to ensure that you validate requests against correct rules. Your average application will likely have many entry points, and a set of rules for each. In this section I will sketch a rules framework that you can use whenever you need to deal with XML.

```
# Establish the baseline for all XML entry points
<Location /api/>
    # Is the Content-Type correct?
    SecRule REQUEST_HEADERS:Content-Type !^text/xml$ \
        "phase:1,t:lowercase,deny,msg:'Invalid Content-Type for XML API'"

    # Activate XML parsing
    SecAction phase:1,t:lowercase,nolog,pass,ctl:requestBodyProcessor=XML
```

```
    # Was the payload successfully parsed?
    SecRule REQBODY_PROCESSOR_ERROR "!@eq 0" \
        "phase:2,t:none,log,block,msg:'Failed to parse request body: …
%{REQBODY_PROCESSOR_ERROR_MSG}'"

    # By default, we assume that XML validation did not take place
    SecAction phase:2,nolog,pass,setvar:TX.xml_validated=0
</Location>

# Entry point One
<Location /api/entryPointOne.php>
    # Validate payload first
    SecRule XML "@validateDTD /path/to/conf/entryPointOne.dtd" \
        "phase:2,deny,msg:'Failed to validate XML against entryPointOne.dtd'"

    # Restrict employee name to known good characters only
    SecRule XML:/employees/employee/name/text() !^[a-zA-Z ]{3,33}$ \
        "phase:2,deny,msg:'Invalid employee name'"

    # Validation was successful
    SecAction phase:2,nolog,pass,setvar:TX.xml_validated
</Location>

# Entry point Two
<Location /api/entryPointTwo.php>
    # Validate payload first
    SecRule XML "@validateDTD /path/to/conf/entryPointTwo.dtd" \
        "phase:2,deny,msg:'Failed to validate XML against entryPointTwo.dtd'"

    # Implement additional restrictions
    # ...

    # Validation was successful
    SecAction phase:2,nolog,pass,setvar:TX.xml_validated
</Location>

# Finally, verify that the entry point was valid
<Location /api/>
    # The xml_validated flag will only be set after a
    # successful validation
    SecRule TX:xml_validated "!@eq 1" \
        "phase:2,deny,msg:'Invalid API entry point'"
</Location>
```

With the above framework, we achieve the following:

1. There is first one `<Location>` section where we establish the baseline for all XML entry points. It is here that we activate XML parsing, but also reject all requests that are

not XML. The assumption is that the `/api/` folder contains only XML entry points. This assumption is usually valid, since API calls do not need any accompanying files (such as embedded images, stylesheet files, etc.).

2. With a further one `<Location>` section per entry point, we ensure that we apply the correct validation rules to each entry point, followed by the per-entry point rules.
3. We finalize the XML rules by adding another global `<Location>` section, where we use one rule that checks if validation was successfully completed. This final check is needed in case a request specifies an unlisted entry point, in which case the `xml_validated` flag will be `0` (set in the first global section).

> **Note**
>
> Remember that configuration merging for the `<Location>` directive works in the same order in which the sections appear in the configuration file. Thus, the rules will be processed in the way they appear in the configuration too.

13 Extending Rule Language

The ModSecurity Rule Language is pretty good at meeting the users' requirements—especially now that it's entering maturity—but sometimes you'll encounter a need for it to do something it cannot. In ModSecurity there is a easy way to extend the rule language by writing C code to extend the rule language. There are three extension points, enabling you to add custom variables, operators and transformation functions.

Because ModSecurity is part of Apache, it does not have to implement its own extension infrastructure: you extend ModSecurity by writing Apache modules! It's a great time-saver if you have previous Apache programming experience. But, even if you don't, finding people who do will be generally easy. After all, Apache is one of the most popular programs ever.

For years, the common way to learn how to write Apache modules was to study existing modules, especially the ones bundled with Apache itself. (My favourite always has been `mod_rewrite`.) These days, however, we have proper documentation, thanks to Nick Kew, who wrote *The Apache Modules Book* (Prentice Hall, 2007). If you are planning to do some serious work you should definitely get Nick's book. For simple efforts, what's in this section should be sufficient.

With or without the book, you should familiarize yourself with the Apache Portable Runtime [http://apr.apache.org] (APR) and the Apache Portable Library Utility [http://apr.apache.org] (APR-Util) libraries, which are the infrastructure and portability library on which Apache is built. Whenever you program an Apache module you have full access to the APR and APR-Util libraries. That is quite handy, as those libraries contain tons of useful functionality.

The remainder of this section will introduce a template module, which you can use as a starting point for your ModSecurity extensions, and then implement three modules, one for each extension point. For the examples I will use the sample code included with ModSecurity and stored in the `apache2/api` subfolder.

Extension Template

I am first going to show you how to create a template module that does not do anything except establishes the infrastructure for our other efforts.

> **Note**
>
> Before you begin you will first need to ensure that you have the ability to compile custom Apache modules. This is the same process as the one where you custom-compile ModSecurity itself. In addition, you will need the source code for the exact version of ModSecurity you are writing extensions for.

The template module is a complete Apache module, which you should be able to compile and install. You can practice with it to ensure that your environment has all the right components for custom Apache module development.

Here is the complete module source code:

```
#include "httpd.h"
#include "http_core.h"
#include "http_config.h"
#include "http_log.h"
#include "http_protocol.h"
#include "ap_config.h
#include "apr_optional.h"

#include "modsecurity.h"

/**
 * This function is just a placeholder in this template.
 */
static int hook_pre_config(apr_pool_t *mp, apr_pool_t *mp_log, apr_pool_t …
*mp_temp) {
    /* Empty for now, but will be used later. */
    return OK;
}

/**
 * Register to be invoked before configuration begins.
 */
static void register_hooks(apr_pool_t *p) {
    ap_hook_pre_config(hook_pre_config, NULL, NULL, APR_HOOK_LAST);
}

/**
 * This structure is used by Apache to determine that a dynamic
```

```
 * library it is loading is a genuine module.
 */
static module AP_MODULE_DECLARE_DATA security_template_module = {
    STANDARD20_MODULE_STUFF,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    register_hooks
};
```

There are three points of interest in the module:

1. The `security_template_module` is a structure used by Apache to verify that the dynamic library is indeed a module. The name here is important and should be unique. You will use the name when you instruct Apache to load the module, later on.
2. The initialization structure `template_module` registers the `register_hook` callback, which is going to be the module's main initialization entry point.
3. The `register_hook` callback registers another callback, `hook_pre_config`, which is invoked every time Apache is reconfigured. This callback doesn't do anything in the template module, but we will add to it later.

You will be compiling the template module using the apxs Apache tool. If it is not in your path, it will be in the `bin/` subfolder of your Apache installation. Assuming you placed the source code in the file called `mod_security_template.c`, to compile the template module invoke:

```
$ apxs -cia -I/path/to/modsecurity/source_code -I/usr/include/libxml2 …
mod_security_template.c
```

> **Note**
>
> On Linux, processes are known to crash when the dynamic libraries they are using change. It is best practice to shut down Apache before adding or removing any of its modules.

The `apxs` command line used above uses 5 switches, which perform the following functions:

1. Compile the module (switch `-c`).
2. Copy the compiled module to the directory where all other Apache modules are stored (switch `-i`).
3. Activate the module by adding the correct `LoadModule` directive to the configuration (switch `-a`).

4. Point to the location of the ModSecurity include files (switch `-I/path/to/modsecurity/source_code`)
5. Point to the location of the libxml2 include files (switch `-I/usr/include/libxml2`)

The activation step will work if you have at least one existing `LoadModule` directive in your configuration. If it did in your case, the last line will say something similar to:

```
[activating module `security_template' in /path/to/apache/conf/httpd.conf].
```

If you have a more elaborate configuration layout and the `apxs` tool cannot find at least one existing `LoadModule` directive in your `httpd.conf`, you will have to activate the module manually. You do that by adding the following one line to the configuration:

```
LoadModule security_template_module modules/mod_security_template.so
```

The first parameter must match the module name used in the source code. You should always place an extension module after the `LoadModule` line that activates ModSecurity. If you don't, ModSecurity might not be able to recognize the newly added function.

If Apache starts with the new `LoadModule` line in the configuration, you've successfully completed this step.

Adding a Transformation Function

Starting from the template module, implementing a new transformation function requires two steps. First you need to implement a single function, which will be called by ModSecurity every time a transformation is needed. All transformation functions (in C) use the following signature:

```
static int reverse(apr_pool_t *mptmp, unsigned char *input,
    long int input_len, char **rval, long int *rval_len)
{
    /* Transformation code here. */

    /* Return 1 if you change the input, 0 if you don't/ */
    return 1;
}
```

You should generally use the same name for the C function as the name you intend to use for the transformation function in ModSecurity. In the signature I used the name `reverse`, which is the same name used in the example below. The five parameters you get are the following:

1. `apr_pool_t *mptp` - APR memory pool you can use to allocate memory from

2. `unsigned char *input` - pointer to the input string you need to transform
3. `long int input_len` - length of the input string
4. `char **rval` - pointer in which to return the output string
5. `long int rval_len` - length of the output string

> **Note**
>
> Remember that ModSecurity does not use NUL-terminated strings. Always use the `input_len` parameter, which contains the input length.

If your transformation always results with an output string that is equal to or shorter than the input string, you should do make your changes in-place, overwriting the input string. By doing that you save on one memory allocation, making your transformation function faster. In this case, the `rval` pointer should point to the input string on return.

If the output can be longer, use the `mptmp` memory pool to allocate from, then point `rval` to the newly allocated memory chunk. The memory you allocate will be deallocated automatically, when ModSecurity clears the temporary memory pool. With any other memory allocation method you would create a memory leak, because deallocation is always manual and you won't have an opportunity to invoke it.

Here's the complete source code of the transformation function example included with ModSecurity:

```
/**
 * This function will be invoked by
 * ModSecurity to transform input.
 */
static int reverse(apr_pool_t *mptmp, unsigned char *input,
    long int input_len, char **rval, long int *rval_len)
{
    /* Transformation functions can choose to do their
     * thing in-place, overwriting the existing content. This
     * is normally possible only if the transformed content
     * is of equal length or shorter.
     *
     * If you need to expand the content use the temporary
     * memory pool mptmp to allocate the space.
     */

    /* Reverse the string in place, but only if it's long enough. */
    if (input_len > 1) {
        long int i = 0;
        long int j = input_len - 1;
        while(i < j) {
```

```
            char c = input[i];
            input[i] = input[j];
            input[j] = c;
            i++;
            j--;
        }
    }

    /* Tell ModSecurity about the content
     * we have generated. In this case we
     * merely point back to the input buffer.
     */
    *rval = (char *)input;
    *rval_len = input_len;

    /* Must return 1 if the content was
     * changed, or 0 otherwise.
     */
    return 1;
}
```

The return value from a transformation function should always be `1` if the content you are returning is different from the content you received on input and `0` otherwise. Even if you placed the output in a newly allocated memory chunk, if it is the same the return code should be `0`. Returning the correct response code will allow ModSecurity to optimize certain things when there are no changes, but you shouldn't worry to much about it. If keeping track of whether you made a change is difficult or expensive (as it might be in the above example), just return `1`.

Now that you have the function, you need to register it with ModSecurity. For that you will use the Apache mechanism called *optional functions*. It's a two-step process:

1. First you ask Apache to find you the registration function, which will have been exported by ModSecurity beforehand.
2. Then you register the new transformation function.

```
/**
 * Register transformation function with ModSecurity.
 */
static int pre_config(apr_pool_t *mp, apr_pool_t *mp_log, apr_pool_t *mp_temp) {
    void (*fn)(const char *name, void *fn);

    /* Look for the registration function
     * exported by ModSecurity.
     */
    fn = APR_RETRIEVE_OPTIONAL_FN(modsec_register_tfn);
```

```
    if (fn) {
        /* Use it to register our new
         * transformation function under the
         * name "reverse".
         */
        fn("reverse", (void *)reverse);
    } else {
        ap_log_error(APLOG_MARK, APLOG_ERR | APLOG_NOERRNO, 0, NULL,
            "mod_tfn_reverse: Unable to find modsec_register_tfn.");
    }

    return OK;
}
```

Once you restart Apache, the new transformation function will be equal to the ones that come with ModSecurity. You should always test your new functionality. For example, add the following rule to the configuration:

```
SecRule ARGS test phase:1,log,deny,t:none,t:reverse
```

Then, if you send a request with parameter `p` whose value set to `tset` (the opposite of `test`, of course), you should get a 403 response in return. The debug log excerpt shows the new reverse transformation function working as expected:

```
[4] Recipe: Invoking rule 9d77ed8; [file "/home/ivanr/apache/conf/httpd.conf"] …
[line "509"].
[5] Rule 9d77ed8: SecRule "ARGS" "@rx test" "phase:1,auditlog,t:none,t:reverse,…
log,deny"
[9] T (0) reverse: "test"
[4] Transformation completed in 56 usec.
[4] Executing operator "rx" with param "test" against ARGS:p.
[9] Target value: "test"
[4] Operator completed in 59 usec.
[4] Rule returned 1.
[9] Match, intercepted -> returning.
[1] Access denied with code 403 (phase 1). Pattern match "test" at ARGS:p. [file …
"/home/ivanr/apache/conf/httpd.conf"] [line "509"]
```

Adding an Operator

Creating new operators is slightly more difficult because two functions are needed: there's an additional (and optional) initialization step, which allows your code to do some work at configure-time and reuse it at run-time. The split of the work sometimes allows for significant performance improvements.

The new operator example adds a new string matching function based on the Boyer-Moore-Horspool algorithm [http://en.wikipedia.org/wiki/Boyer%E2%80%93Moore%E2%80%93Horspool_algorithm]. I will not show here the code for the algorithm itself, but we will assume that we already have the following two functions implemented:

```
static void initBoyerMooreHorspool(const char *pattern, int patlength,
    int *bm_badcharacter_array);

static int BoyerMooreHorspool(const char *pattern, int patlength,
    const char *text, int textlen, int *bm_badcharacter_array);
```

If you are curious, of course, you can always look at the implementation at the end of the `mod_op_strstr.c` file (included with ModSecurity). The string matching algorithm does require initialization, so we will be using both steps.

The initialization code is as follows:

```
/**
 * Operator parameter initialisation entry point.
 */
static int op_strstr_init(msre_rule *rule, char **error_msg) {
    /* Operator initialisation function will be called once per
     * statement where operator is used. It is meant to be used
     * to check the parameters to see whether they are present
     * and if they are in the correct format.
     */

    /* In this example we just look for a simple non-empty parameter. */
    if ((rule->op_param == NULL)||(strlen(rule->op_param) == 0)) {
        *error_msg = apr_psprintf(rule->ruleset->mp, "Missing parameter …
for operator 'strstr'.");
        return 0; /* ERROR */
    }

    /* If you need to transform the data in the parameter into something
     * else you should do that here. Simply create a new structure to hold
     * the transformed data and place the pointer to it into rule->op_param_data.
     * You will have access to this pointer later on.
     */
    rule->op_param_data = apr_pcalloc(rule->ruleset->mp, …
ALPHABET_SIZE * sizeof(int));
    initBoyerMooreHorspool(rule->op_param, strlen(rule->op_param), …
(int *)rule->op_param_data);

    /* OK */
    return 1;
}
```

Unlike with the previous example (the transformation function), here we get to work with ModSecurity structures directly. The first parameter of the operator initialization is a pointer to the `msre_rule` structure (full definition in `re.h`). There are two fields in this structure that you will want to use:

- `op_param` - a NUL-terminated string that may contain a parameter for your operator.
- `op_param_data` - a generic pointer for your operators' use.

The idea is that you will check the parameter available in `op_param` and do something with it, then perform the initialization work and store a pointer to the results in `op_param_data`. When your operator is invoked at run-time it will have access to the same `msre_rule` structure, and thus to `op_param_data`.

- If you need to allocate memory, use the memory pool in `rule->ruleset->mp`, as in the example.
- If your initialization fails, generate an error string, store it in `error_msg` (the second function parameter), and return a zero.

The operator execution code is equally simple:

```
/**
 * Operator execution entry point.
 */
static int op_strstr_exec(modsec_rec *msr, msre_rule *rule, msre_var *var, …
char **error_msg) {
    /* Here we need to inspect the contents of the supplied variable. */

    /* In a general case it is possible for the value
     * to be NULL. What you need to do in this case
     * depends on your operator. In this example we return
     * a "no match" response.
     */
    if (var->value == NULL) return 0; /* No match. */

    /* Another thing to note is that variables are not C strings,
     * meaning the NULL byte is not used to determine the end
     * of the string. Variable length var->value_len should be
     * used for this purpose.
     */

    if (BoyerMooreHorspool(rule->op_param, strlen(rule->op_param),
        var->value, var->value_len, (int *)rule->op_param_data) >= 0)
    {
        return 1; /* Match. */
    }
```

```
    return 0; /* No match. */
}
```

This time you will receive 4 parameters:

1. `modsec_rec *msr` - the structure where all transaction data is stored.
2. `msre_rule *rule` - the same rule structure you received in the initialization phase
3. `msre_var *var` - the variable structure, which holds the data the operators needs to inspect.
4. `char **error_msg` - an error message pointer, which needs to contain an error message on error.

The data you need to inspect is stored in a `msre_var` instance, which has the following layout:

```
struct msre_var {
    const char              *name;
    const char              *value;
    unsigned int             value_len;
    const char              *param;
    const void              *param_data;
    msre_var_metadata       *metadata;
    msc_regex_t             *param_regex;
    unsigned int             is_negated;
    unsigned int             is_counting;
};
```

Although it looks complex, you only need to be concerned with two fields:

- `const char *value` - pointer to the variable the operator needs to inspect.
- `unsigned int value_len` - the length of the variable.

After you inspect the variable, return `0` if there is no match, and `1` if there is.

The operator registration step is conceptually identical to that used for transformation functions, except that you use the `modsec_register_operator` optional function:

```
static int hook_pre_config(apr_pool_t *mp, apr_pool_t *mp_log, …
apr_pool_t *mp_temp) {
    void (*fn)(const char *name, void *fn_init, void *fn_exec);

    /* Look for the registration function
     * exported by ModSecurity.
     */
    fn = APR_RETRIEVE_OPTIONAL_FN(modsec_register_operator);
    if (fn) {
        /* Use it to register our new
         * transformation function under the
         * name "reverse".
```

```
         */
        fn("strstr", (void *)op_strstr_init, (void *)op_strstr_exec);
    } else {
        ap_log_error(APLOG_MARK, APLOG_ERR | APLOG_NOERRNO, 0, NULL,
            "mod_op_strstr: Unable to find modsec_register_operator.");
    }

    return OK;
}
```

Adding a Variable

To generate new variables you typically need to implement one function call. The example that comes with ModSecurity is actually split across three functions, but that's done for code reuse. Here's the simplified code:

```
static int var_remote_addr_port_generate(modsec_rec *msr, msre_var *var,
    msre_rule *rule, apr_table_t *vartab, apr_pool_t *mptmp)
{

    msre_var *rvar = NULL;

    if (value == NULL) return 0;

    /* Generate new variable. */
    rvar = apr_pmemdup(mptmp, var, sizeof(msre_var));
    rvar->value = apr_psprintf(mptmp, "%s:%d", msr->remote_addr, msr->remote_port);;
    rvar->value_len = strlen(rvar->value);

    /* Add variable to the collection. */
    apr_table_addn(vartab, rvar->name, (void *)rvar);

    return 1;
}
```

The following parameters are provided:

1. `modsec_rec *msr` - the structure where all transaction data is stored.
2. `msre_var *var` - variable template.
3. `apr_table_t *vartab` - the collection used to store the variables being prepared for inspection.
4. `apr_pool_t *mptmp` - the memory pool from which you can allocate memory.

Creating new variables is a four-step process:

1. First create the variable data. How you do that depends on the nature of the data, but it can be as easy as using a single `apr_sprintf()` call (as in the example).
2. Create a new `msre_var` structure, duplicating from the one already provided in `var`, and populate the `value` and `value_len` fields.
3. Using `apr_table_addn()`, add the newly created `msre_var` structure to the `vartab` collection.
4. Return `1` to indicate that you've added one variable to the collection. If you create more than one variable (by just repeating steps 1-3, keep track of how many new variables there are and return it at the end of the function).

Variable registration is slightly more involved, but only because you need to help ModSecurity do most of the configuration work for you.

```
static int hook_pre_config(apr_pool_t *mp, apr_pool_t *mp_log, …
apr_pool_t *mp_temp) {
    void (*register_fn)(const char *name, unsigned int type,
                        unsigned int argc_min, unsigned int argc_max,
                        void *fn_validate, void *fn_generate,
                        unsigned int is_cacheable, unsigned int availability);

    /* Look for the registration function
     * exported by ModSecurity.
     */
    register_fn = APR_RETRIEVE_OPTIONAL_FN(modsec_register_variable);
    if (register_fn) {
        /* Use it to register our new
         * variable under the
         * name "REMOTE_ADDR_PORT".
         */
        register_fn(
            "REMOTE_ADDR_PORT",
            VAR_SIMPLE,
            0, 0,
            NULL,
            var_remote_addr_port_generate,
            VAR_DONT_CACHE,
            PHASE_REQUEST_HEADERS
        );
    }  else {
        ap_log_error(APLOG_MARK, APLOG_ERR | APLOG_NOERRNO, 0, NULL,
            "mod_var_remote_addr_port: Unable to find modsec_register_variable.");
    }

    return OK;
}
```

To register a variable you need to use 8 parameters, but apart from that, the registration process does not hold any surprises:

1. `const char *name` - variable name.
2. `unsigned int type` - variable type; use `VAR_SIMPLE` to indicate that you will return only one value, or `VAR_LIST` to indicate the possibly of returning multiple values.
3. `unsigned int argc_min` - variable parameter definition; use `0` if you don't need to use a parameter, or `1` if you do.
4. `unsigned int argc_max` - variable parameter definition; use `0` if you do not allow a parameter, or `1` if you do.
5. `void *fn_validate` - optional pointer to the parameter validation function.
6. `void *fn_generate` - pointer to the generation function.
7. `unsigned int is_cacheable` - is the variable cacheable? If generating the variable is expensive and the value is not likely to change during the duration of a transaction, use `VAR_CACHE`. Otherwise, use `VAR_DONT_CACHE`.
8. `unsigned int availability` - the phase in which the variable becomes available: `PHASE_REQUEST_HEADERS`, `PHASE_REQUEST_BODY`, `PHASE_RESPONSE_HEADERS`, `PHASE_RESPONSE_BODY` or `PHASE_LOGGING`. ModSecurity should use this value to ensure the variable isn't referenced in the rules before it is available. (I say should because ModSecurity does not do that at the moment.)

As you know, in ModSecurity variables can have parameters. For example, you use `ARGS:p` to request the parameter named `p`, and `ARGS:/^p/` to request all the parameters that start with `p`. If you allow parameters for your variables, the single parameter will be placed in `var->param`. How you interpret the parameter depends on the nature of the variable. For inspiration you can look up the `var_args_generate()` function in `re_variables.c`, which implements the `ARGS` collection.

Finally, if you think you can speed-up variable retrieval by using configure-time initialization, supply a separate validation function when you register your variable. For example, you could use the following template:

```
static char *var_generic_list_validate(msre_ruleset *ruleset, msre_var *var) {
    /* Is it OK if there's no parameter provided? Return NULL if
     * it is. If you require a parameter and you correctly registered
     * the variable, your validation function will never be invoked.
     */
    if (var->param == NULL) return NULL;

    /* Validate the value in var->param. */
    // ...
```

```
    /* Perform your initialization work. */
    // ...

    /* Store initialization data for subsequent retrieval. */
    var->param_data = my_opaque_pointer;

    /* No error. */
    return NULL;
}
```

Part II: Reference Documentation

14 Reference Manual

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`, `LocationMatch`, `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. The first version to use a given directive is given in the Version sections below.

SecAction

Description: Unconditionally processes the action list it receives as the first and only parameter. The syntax of the parameter is identical to that of the third parameter of `SecRule`.

Syntax: `SecAction action1,action2,action3,...`

Scope: Any

Version: 2.0.0

`SecAction` is best used when you need something to always happen, irrespective of any transaction condition. The actions specified in `SecRule` are conditional based on data inspection of the request/response. This directive is commonly used to initialize persistent collections using the `initcol` action. For example:

```
SecAction nolog,phase:1,initcol:RESOURCE=%{REQUEST_FILENAME}
```

SecArgumentSeparator

Description: Specifies which character to use as separator for `application/x-www-form-urlencoded` content.

Syntax: `SecArgumentSeparator character`

Default: `&`

Scope: Main

Version: 2.0.0

This directive is needed if a backend web application is using a non-standard argument separator. Applications are sometimes (very rarely) written to use a semicolon separator. You should not change the default setting unless you establish that the application you are working with requires a different separator. If this directive is not set properly for each web application, then ModSecurity will not be able to parse the arguments appropriately and the effectiveness of the rule matching will be significantly decreased.

SecAuditEngine

Description: Configures the audit logging engine.

Syntax: `SecAuditEngine On|Off|RelevantOnly`

Default: `Off`

Scope: Any

Version: 2.0.0

The `SecAuditEngine` directive is used to configure the audit engine, which logs complete transactions.

> **Note**
>
> If you need to change the audit log engine configuration on per-transaction basis (e.g., in response to some transaction data), use the `ctl` action.

The following example demonstrates how `SecAuditEngine` is used:

```
SecAuditEngine RelevantOnly
SecAuditLog logs/audit/audit.log
SecAuditLogParts ABCFHZ
SecAuditLogType concurrent
SecAuditLogStorageDir logs/audit
SecAuditLogRelevantStatus ^(?:5|4\d[^4])
```

The possible values for the audig log engine are as follows:

- `On` - log all transactions
- `Off` - do not log any transactions

- `RelevantOnly` - only the log transactions that have triggered a warning or an error, or have a status code that is considered to be relevant (as determined by the `SecAuditLogRelevantStatus` directive)

SecAuditLog

Description: Defines the path to the main audit log file (serial logging format) or the concurrent logging index file (concurrent logging format). When used in combination with mlogc, this directive defines the mlogc location and command line (when concurrent logging is used).

Syntax: `SecAuditLog /path/to/audit.log`

Scope: Any

Version: 2.0.0

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 to send your audit log data off to a remote server you will need to deploy the ModSecurity Log Collector (mlogc), like this:

```
SecAuditLog "|/path/to/mlogc /path/to/mlogc.conf"
```

> **Note**
>
> This audit log file is opened 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.

SecAuditLog2

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

Syntax: `SecAuditLog2 /path/to/audit.log`

Scope: Any

Version: 2.1.2

The purpose of `SecAuditLog2` is to make logging to two remote servers possible, which is typically achieved by running two instances of the mlogc tool, each with a different configuration (in addition, one of the instances will need to be instructed not to delete any files

it submits). This directive can be used only if `SecAuditLog` was previously configured and only if concurrent logging format is used.

SecAuditLogDirMode

Description: Configures the mode (permissions) of any directories created for the concurrent audit logs, using an octal mode value as parameter (as used in chmod).

Syntax: `SecAuditLogDirMode octal_mode|"default"`

Default: `0600`

Scope: Any

Version: 2.5.10

The default mode for new audit log directories (`0600`) only grants read/write access to the directory owner (typically the account under which Apache is running). If access from other accounts is needed (e.g., for use with `mpm-itk`), then you may use this directive to gran additional read and/or write privileges. You should use this directive with caution to avoid exposing potentially sensitive data to unauthorized users. Using the value `default` as parameter reverts the configuration back to the default setting. This feature is not available on operating systems not supporting octal file modes.

Example:

```
SecAuditLogDirMode 02750
```

> **Note**
>
> The process umask may still limit the mode if it is being more restrictive than the mode set using this directive.

SecAuditLogFileMode

Description: Configures the mode (permissions) of any files created for concurrent audit logs using an octal mode (as used in chmod). See `SecAuditLogDirMode` for controlling the mode of created audit log directories.

Syntax: `SecAuditLogFileMode octal_mode|"default"`

Example Usage: `SecAuditLogFileMode 00640`

Scope: Any

Version: 2.5.10

Dependencies/Notes: This feature is not available on operating systems not supporting octal file modes. The default mode (0600) only grants read/write access to the account writing the file. If access from another account is needed (using mpm-itk is a good example), then this directive may be required. However, 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.

> **Note**
>
> The process umask may still limit the mode if it is being more restrictive than the mode set using this directive.

SecAuditLogParts

Description: Defines which part of each transaction are going to be recorded in audit log. Each part is assigned a single letter. If a letter appears in the list then the equivalent part of each transactions will be recorded. See below for the list of all parts.

Syntax: `SecAuditLogParts PARTS`

Example Usage: `SecAuditLogParts ABCFHZ`

Scope: Any

Version: 2.0.0

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`.

> **Note**
>
> Please refer to the ModSecurity Data Formats document for a detailed description of every available part.

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). Interme-

diary 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.
- `K` - This part contains a full list of every rule that matched (one per line) in the order they were matched. The rules are fully qualified and will thus show inherited actions and default operators. Supported as of v2.5.0
- `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 ^(?:5|4\d[^4])`

Scope: Any

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

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.

SecCacheTransformations (Deprecated/Experimental)

Description: Controls caching of transformations. Caching is off by default starting with 2.5.6, when it was deprecated and downgraded back to experimental.

Syntax: `SecCacheTransformations On|Off [options]`

Example Usage: `SecCacheTransformations On "minlen:64,maxlen:0"`

Scope: Any

Version: 2.5.0

Dependencies/Notes: N/A

First parameter:

- `On` - cache transformations (per transaction, per phase) allowing identical transformations to be performed only once. (default)
- `Off` - do not cache any transformations, forcing all transformations to be performed for each rule executed.

The following options are allowed (comma separated):

- `incremental:on|off` - enabling this option will cache every transformation instead of just the final transformation. (default: off)
- `maxitems:N` - do not allow more than N transformations to be cached. The cache will then be disabled. A zero value is interpreted as "unlimited". This option may be useful to limit caching for a form with a large number of ARGS. (default: 512)
- `minlen:N` - do not cache the transformation if the value's length is less than N bytes. (default: 32)
- `maxlen:N` - do not cache the transformation if the value's length is more than N bytes. A zero value is interpreted as "unlimited". (default: 1024)

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`

Scope: Main

Version: 2.0.0

Dependencies/Notes: This feature is not available on Windows builds. 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.builds. Some problems you might encounter with more complex setups:

1. DNS lookups do not work (this is because this feature requires a shared library that is loaded on demand, after chroot takes place).
2. You cannot send email from PHP because it uses sendmail and sendmail is outside the jail.
3. In some cases Apache graceful (reload) no longer works.

You should be aware that the internal chroot feature might not be 100% reliable. 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. A module, working from within Apache, can do things that make it easy to break out of the jail. 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. Consider what your options are and make your own decision.

SecComponentSignature

Description: Appends component signature to the ModSecurity signature.

Syntax: `SecComponentSignature "COMPONENT_NAME/X.Y.Z (COMMENT)"`

Example usage: `SecComponentSignature "Core Rules/1.2.3"`

Scope: Main

Version: 2.5.0

Dependencies/Notes: This directive should be used to make the presence of significant ModSecurity components known. The entire signature will be recorded in transaction audit log. It should be used by ModSecurity module and rule set writers to make debugging easier.

SecContentInjection

Description: Enables content injection using actions `append` and `prepend`.

Syntax: `SecContentInjection (On|Off)`

Example Usage: `SecContentInjection On`

Scope: Any

Version: 2.5.0

Dependencies/Notes: N/A

SecCookieFormat

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

Syntax: `SecCookieFormat 0|1`

Example Usage: `SecCookieFormat 0`

Scope: Any

Version: 2.0.0

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`

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`

Scope: Any

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

Dependencies/Notes: Rules following a `SecDefaultAction` directive will inherit this setting unless a specific action is specified for an individual rule or until another `SecDefaultAction` is specified. Take special note that in the logging disruptive actions are not allowed, but this can inadvertently be inherited using a disruptive action in `SecDefaultAction`.

The default value is minimal (differing from previous versions):

```
SecDefaultAction phase:2,log,auditlog,pass
```

> **Note**
>
> `SecDefaultAction` must specify a disruptive action and a processing phase and cannot contain metadata actions.

> **Warning**
>
> `SecDefaultAction` is *not* inherited across configuration contexts. (For an example of why this may be a problem for you, read the following ModSecurity Blog entry http://blog.modsecurity.org/2008/07/modsecurity-tri.html).

SecGeoLookupDb

Description: Defines the path to the geographical database file.

Syntax: `SecGeoLookupDb /path/to/db`

Example Usage: `SecGeoLookupDb /usr/local/geo/data/GeoLiteCity.dat`

Scope: Any

Version: 2.5.0

Dependencies/Notes: Check out `maxmind.com` for free database files.

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`

Scope: Main

Version: 2.0.0

Dependencies/Notes: By default httpd-guardian will defend against clients that send more than 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 Snort-Sam (http://www.snortsam.net). Assuming httpd-guardian is already configured (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
```

SecMarker

Description: Adds a fixed rule marker in the ruleset to be used as a target in a `skipAfter` action. A `SecMarker` directive essentially creates a rule that does nothing and whose only purpose it to carry the given ID.

Syntax: `SecMarker ID`

Example Usage: `SecMarker 9999`

Scope: Any

Version: 2.5.0

Dependencies/Notes: None

```
SecRule REQUEST_URI "^/$" \
    "chain,t:none,t:urlDecode,t:lowercase,t:normalizePath,skipAfter:99"
SecRule REMOTE_ADDR "^127\.0\.0\.1$" "chain"
SecRule REQUEST_HEADERS:User-Agent \
    "^Apache \(internal dummy connection\)$" "t:none"
SecRule &REQUEST_HEADERS:Host "@eq 0" \
    "deny,log,status:400,id:08,severity:4,msg:'Missing a Host Header'"
SecRule &REQUEST_HEADERS:Accept "@eq 0" \
    "log,deny,log,status:400,id:15,msg:'Request Missing an Accept Header'"

SecMarker 99
```

SecPdfProtect (Obsolete)

Description: Enables the PDF XSS protection functionality. Once enabled access to PDF files is tracked. Direct access attempts are redirected to links that contain one-time tokens. Requests with valid tokens are allowed through unmodified. Requests with invalid tokens are

also allowed through but with forced download of the PDF files. This implementation uses response headers to detect PDF files and thus can be used with dynamically generated PDF files that do not have the `.pdf` extension in the request URI.

Syntax: `SecPdfProtect On|Off`

Example Usage: `SecPdfProtect On`

Scope: Any

Version: 2.5.0; removed in 2.6.0

Dependencies/Notes: None

SecPdfProtectMethod (Obsolete)

Description: Configure desired protection method to be used when requests for PDF files are detected. Possible values are `TokenRedirection` and `ForcedDownload`. The token redirection approach will attempt to redirect with tokens where possible. This allows PDF files to continue to be opened inline but only works for GET requests. Forced download always causes PDF files to be delivered as opaque binaries and attachments. The latter will always be used for non-GET requests. Forced download is considered to be more secure but may cause usability problems for users ("This PDF won't open anymore!").

Syntax: `SecPdfProtectMethod method`

Example Usage: `SecPdfProtectMethod TokenRedirection`

Scope: Any

Version: 2.5.0; removed in 2.6.0

Dependencies/Notes: None

Default: `TokenRedirection`

SecPdfProtectSecret (Obsolete)

Description: Defines the secret that will be used to construct one-time tokens. You should use a reasonably long value for the secret (e.g. 16 characters is good). Once selected the secret should not be changed as it will break the tokens that were sent prior to change. But it's not a big deal even if you change it. It will just force download of PDF files with tokens that were issued in the last few seconds.

Syntax: `SecPdfProtectSecret secret`

Example Usage: `SecPdfProtectSecret MyRandomSecretString`

Scope: Any

Version: 2.5.0; removed in 2.6.0

Dependencies/Notes: None

SecPdfProtectTimeout (Obsolete)

Description: Defines the token timeout. After token expires it can no longer be used to allow access to PDF file. Request will be allowed through but the PDF will be delivered as attachment.

Syntax: `SecPdfProtectTimeout timeout`

Example Usage: `SecPdfProtectTimeout 10`

Scope: Any

Version: 2.5.0; removed in 2.6.0

Dependencies/Notes: None

Default: `10`

SecPdfProtectTokenName (Obsolete)

Description: Defines the name of the token. The only reason you would want to change the name of the token is if you wanted to hide the fact you are running ModSecurity. It's a good reason but it won't really help as the adversary can look into the algorithm used for PDF protection and figure it out anyway. It does raise the bar slightly so go ahead if you want to.

Syntax: `SecPdfProtectTokenName name`

Example Usage: `SecPdfProtectTokenName PDFTOKEN`

Scope: Any

Version: 2.5.0; removed in 2.6.0

Dependencies/Notes: None

Default: `PDFTOKEN`

SecRequestBodyAccess

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

Syntax: `SecRequestBodyAccess On|Off`

Example Usage: `SecRequestBodyAccess On`

Scope: Any

Version: 2.0.0

Dependencies/Notes: This directive is required if you plan to inspect `POST_PAYLOAD`. 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`

Scope: Any

Version: 2.0.0

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.

SecRequestBodyNoFilesLimit

Description: Configures the maximum request body size ModSecurity will accept for buffering, excluding the size of files being transported in the request. This directive comes handy to further reduce susceptibility to DoS attacks when someone is sending request bodies of very large sizes. Web applications that require file uploads must configure `SecRequestBodyLimit` to a high value. Since large files are streamed to disk file uploads will not increase memory consumption. However, it's still possible for someone to take advantage of a large request body limit and send non-upload requests with large body sizes. This directive eliminates that loophole.

Syntax: `SecRequestBodyNoFilesLimit NUMBER_IN_BYTES`

Example Usage: `SecRequestBodyLimit 131072`

Scope: Any

Version: 2.5.0

Dependencies/Notes: 1 MB (1048576 bytes) is the default setting. This value is very conservative. For most applications you should be able to reduce it down to 128 KB or lower. Anything over the 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`

Scope: Any

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

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
```

SecResponseBodyLimitAction

Description: Controls what happens once a response body limit, configured with `SecResponseBodyLimit`, is encountered. By default ModSecurity will reject a response body that is longer than specified. Some web sites, however, will produce very long responses making it difficult to come up with a reasonable limit. Such sites would have to raise the limit significantly to function properly defying the purpose of having the limit in the first place (to control memory consumption). With the ability to choose what happens once a limit is reached site administrators can choose to inspect only the first part of the response, the part that can fit into the desired limit, and let the rest through. Some could argue that allowing parts of responses to go uninspected is a weakness. This is true in theory but only applies to cases where the attacker controls the output (e.g. can make it arbitrary long). In such cases, however, it is not possible to prevent leakage anyway. The attacker could compress, obfuscate, or even encrypt data before it is sent back, and therefore bypass any monitoring device.

Syntax: `SecResponseBodyLimitAction Reject|ProcessPartial`

Example Usage: `SecResponseBodyLimitAction ProcessPartial`

Scope: Any

Version: 2.5.0

Dependencies/Notes: None

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`

Scope: Any

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

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" \`

`"phase:1,t:none,t:urlDecode,t:lowercase,t:normalizePath"`

Scope: Any

Version: 2.0.0

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 ARGS dirty
```

Each rule can specify one or more variables:

```
SecRule ARGS|REQUEST_HEADERS:User-Agent 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**
>
> 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.

Collections

A variable can contain one or many pieces of data, depending on the nature of the variable and the way it is used. We've seen examples of both approaches in the previous section. When a variable can contain more than one value we refer to it as a *collection*.

Collections are always expanded before a rule is run. For example, the following rule:

```
SecRule ARGS dirty
```

will be expanded to:

```
SecRule ARGS:p dirty
```

```
SecRule ARGS:q dirty
```

in a requests that has only two parameters, named `p` and `q`.

Collections come in several flavours:

Read-only
: Created at runtime using transaction data. For example: `ARGS` (contains a list of all request parameter values) and `REQUEST_HEADERS` (contains a list of all request header values).

Transient Read/Write
: The `TX` collection is created (empty) for every transaction. Rules can read from it and write to it (using the `setvar` action, for example), but the information stored in this collection will not survive the end of transaction.

Persistent Read/Write
: There are several collections that can be written to, but which are persisted to the storage backend. These collections are used to track clients across transactions. Examples of collections that fall into this type are `IP`, `SESSION` and `USER`.

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` (regular expression) operator. You can also explicitly specify the operator you want to use by using `@`, followed by the name of an operator, at the beginning of the second `SecRule` parameter:

```
SecRule ARGS "@rx dirty"
```

Note how we had to use double quotes to delimit the second rule parameter. This is because the second parameter now has 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$"
```

Operator negation

Operator results can be negated by using an exclamation mark at the beginning of the second parameter. The following rule matches if the word `dirty` does *not* appear in the `User-Agent` request header:

```
SecRule REQUEST_HEADERS:User-Agent !dirty
```

You can use the exclamation mark in combination with any parameter. If you do, the exclamation mark needs to go first, followed by the explicit operator reference. The following rule has the same effect as the previous example:

```
SecRule REQUEST_HEADERS:User-Agent "!@rx dirty"
```

If you need to use negation in a rule that is going to be applied to several variables then it may not be immediately clear what will happen. Consider the following example:

```
SecRule ARGS:p|ARGS:q !dirty
```

The above rule is identical to:

```
SecRule ARGS:p !dirty
SecRule ARGS:q !dirty
```

> **Warning**
>
> Negation is applied to operations against individual operations, not agains the entire variable list.

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`

Scope: Any

Version: 2.0.0

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.

```
SecRuleEngine 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.

> **Note**
>
> Configuration contexts are an Apache concept. Directives `<Directory>`, `<Files>`, `<Location>` and `<VirtualHost>` are all used to create configuration contexts. For more information please go to the Apache documentation section Configuration Sections [http://httpd.apache.org/docs/2.0/sections.html].

SecRuleEngine

Description: Configures the rules engine.

Syntax: `SecRuleEngine On|Off|DetectionOnly`

Example Usage: `SecRuleEngine On`

Scope: Any

Version: 2.0.0

Dependencies/Notes: This directive can also be controlled 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: `SecRuleUpdateActionById RULEID ACTIONLIST`

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

Scope: Any

Version: 2.0.0

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"`

Scope: Any

Version: 2.0.0

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).

`SecRuleScript` (Experimental)

Description: This directive creates a special rule that executes a Lua script to decide whether to match or not. The main difference from `SecRule` is that there are no targets nor operators. The script can fetch any variable from the ModSecurity context and use any (Lua) operator to test them. The second optional parameter is the list of actions whose meaning is identical to that of `SecRule`.

Syntax: `SecRuleScript /path/to/script.lua [ACTIONS]`

Example Usage: `SecRuleScript "/path/to/file.lua" "block"`

Scope: Any

Version: 2.5.0

Dependencies/Notes: None

> **Note**
>
> All Lua scripts are compiled at configuration time and cached in memory. To reload scripts you must reload the entire ModSecurity configuration by restarting Apache.

Example script:

```
-- Your script must define the main entry
-- point, as below.
function main()
    -- Log something at level 1. Normally you shouldn't be
    -- logging anything, especially not at level 1, but this is
    -- just to show you can. Useful for debugging.
    m.log(1, "Hello world!");

    -- Retrieve one variable.
    local var1 = m.getvar("REMOTE_ADDR");

    -- Retrieve one variable, applying one transformation function.
    -- The second parameter is a string.
    local var2 = m.getvar("ARGS", "lowercase");

    -- Retrieve one variable, applying several transformation functions.
```

```
    -- The second parameter is now a list. You should note that m.getvar()
    -- requires the use of comma to separate collection names from
    -- variable names. This is because only one variable is returned.
    local var3 = m.getvar("ARGS.p", { "lowercase", "compressWhitespace" } );

    -- If you want this rule to match return a string
    -- containing the error message. The message must contain the name
    -- of the variable where the problem is located.
    -- return "Variable ARGS:p looks suspicious!"

    -- Otherwise, simply return nil.
    return nil;
end
```

In this first example we were only retrieving one variable at the time. In this case the name of the variable is known to you. In many cases, however, you will want to examine variables whose names you won't know in advance, for example script parameters.

Example showing use of `m.getvars()` to retrieve many variables at once:

```
function main()
    -- Retrieve script parameters.
    local d = m.getvars("ARGS", { "lowercase", "htmlEntityDecode" } );

    -- Loop through the paramters.
    for i = 1, #d do
        -- Examine parameter value.
        if (string.find(d[i].value, "<script")) then
            -- Always specify the name of the variable where the
            -- problem is located in the error message.
            return ("Suspected XSS in variable " .. d[i].name .. ".");
        end
    end

    -- Nothing wrong found.
    return nil;
end
```

> **Note**
>
> Go to http://www.lua.org/ to find more about the Lua programming language. The reference manual too is available online, at http://www.lua.org/manual/5.1/.

> **Note**
>
> Lua support is marked as *experimental* as the way the progamming interface may continue to evolve while we are working for the best implementation style. Any user input into the programming interface is appreciated.

SecRuleUpdateActionById

Description: Updates the action list of the specified rule.

Syntax: `SecRuleRemoveById RULEID ACTIONLIST`

Example Usage: `SecRuleUpdateActionById 12345 deny,status:403`

Scope: Any

Version: 2.5.0

Dependencies/Notes: This directive merges the specified action list with the rule's action list. There are two limitations. The rule ID cannot be changed, nor can the phase. Further note that actions that may be specified multiple times are appended to the original.

```
SecAction \
  "t:lowercase,phase:2,id:12345,pass,msg:'The Message',log,auditlog"
SecRuleUpdateActionById 12345 "t:compressWhitespace,deny,status:403,msg:'A new message'
```

The example above will cause the rule to be executed as if it was specified as follows:

```
SecAction \
  "t:lowercase,phase:2,id:12345,log,auditlog,t:compressWhitespace,deny,status:403,msg:'A new message'"
```

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"`

Scope: Main

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

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`

Scope: Any

Version: 2.0.0

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 mode (as used in chmod).

Syntax: `SecUploadFileMode octal_mode|"default"`

Example Usage: `SecUploadFileMode 0640`

Scope: Any

Version: 2.1.6

Dependencies/Notes: This feature is not available on operating systems not supporting octal file modes. The default mode (0600) only grants read/write access to the account writing the file. If access from another account is needed (using clamd is a good example), then this directive may be required. However, 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.

> **Note**
>
> The process umask may still limit the mode if it is being more restrictive than the mode set using this directive.

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`

Scope: Any

Version: 2.0.0

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"`

Scope: Any

Version: 2.0.0

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 conjunction 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.

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). To look at only the query string or body arguments, see the `ARGS_GET` and `ARGS_POST` collections.

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**
>
> Using `ARGS:p` will not result in any invocations against the operator if argument p does not exist.
>
> In ModSecurity 1.X, the `ARGS` variable stood for `QUERY_STRING` + `POST_PAYLOAD`, whereas now it expands 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,t:none,t:lowercase,t:normalizePath"
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,t:none,t:lowercase,t:normalizePath"
SecRule ARGS_NAMES "!^(p|a)$" "t:none,t:lowercase"
```

ARGS_GET

ARGS_GET is similar to ARGS, but only contains arguments from the query string.

ARGS_GET_NAMES

ARGS_GET_NAMES is similar to ARGS_NAMES, but only contains argument names from the query string.

ARGS_POST

ARGS_POST is similar to ARGS, but only contains arguments from the POST body.

ARGS_POST_NAMES

ARGS_POST_NAMES is similar to ARGS_NAMES, but only contains argument names from the POST body.

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 inspect the REQUEST_HEADERS:Authorization header.

DURATION

Contains the number of milliseconds elapsed since the beginning of the current transaction. Available starting with 2.6.0.

ENV

Collection, requires a single parameter (after colon). 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"
```

GEO

`GEO` is a collection populated by the results of the last `@geoLookup` operator. The collection can be used to match geographical fields looked from an IP address or hostname.

Available since ModSecurity 2.5.0.

Fields:

- *COUNTRY_CODE:* Two character country code. EX: US, GB, etc.
- *COUNTRY_CODE3:* Up to three character country code.
- *COUNTRY_NAME:* The full country name.
- *COUNTRY_CONTINENT:* The two character continent that the country is located. EX: EU
- *REGION:* The two character region. For US, this is state. For Canada, providence, etc.
- *CITY:* The city name if supported by the database.
- *POSTAL_CODE:* The postal code if supported by the database.
- *LATITUDE:* The latitude if supported by the database.
- *LONGITUDE:* The longitude if supported by the database.
- *DMA_CODE:* The metropolitan area code if supported by the database. (US only)
- *AREA_CODE:* The phone system area code. (US only)

Example:

```
SecGeoLookupDb /usr/local/geo/data/GeoLiteCity.dat
...
SecRule REMOTE_ADDR "@geoLookup" "chain,drop,msg:'Non-GB IP address'"
SecRule GEO:COUNTRY_CODE "!@streq GB"
```

HIGHEST_SEVERITY

This variable holds the highest severity of any rules that have matched so far. Severities are numeric values and thus can be used with comparison operators such as `@lt`, etc.

> **Note**
>
> Higher severities have a lower numeric value.
>
> A value of 255 indicates no severity has been set.

```
SecRule HIGHEST_SEVERITY "@le 2" "phase:2,deny,status:500,msg:'severity %{HIGHEST_SEVERITY}'"
```

MATCHED_VAR

This variable holds the value of the variable that was matched against. It is similar to the TX:0, except it can be used for all operators and does not require that the `capture` action be specified.

```
SecRule ARGS pattern chain,deny
...
SecRule MATCHED_VAR "further scrutiny"
```

MATCHED_VAR_NAME

This variable holds the full name of the variable that was matched against.

```
SecRule ARGS pattern setvar:tx.mymatch=%{MATCHED_VAR_NAME}
...
SecRule TX:MYMATCH "@eq ARGS:param" deny
```

MODSEC_BUILD

This variable holds the ModSecurity build number. This variable is intended to be used to check the build number prior to using a feature that is available only in a certain build. Example:

```
SecRule MODSEC_BUILD "!@ge 02050102" skipAfter:12345
SecRule ARGS "@pm some key words" id:12345,deny,status:500
```

MULTIPART_CRLF_LF_LINES

This flag variable will be set to 1 whenever a multi-part request uses mixed line terminators. The multipart/form-data RFC requires CRLF sequence to be used to terminate lines. Since some client implementations 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 multi-part flag variable individually, avoiding 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 MULTIPART_INVALID_QUOTING. 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}, \
IQ %{MULTIPART_INVALID_QUOTING}'"
```

The multipart/form-data parser was 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 way 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).

Example:

```
SecRule REQUEST_BASENAME "^login\.php$" phase:2,t:none,t:lowercase
```

> **Note**
>
> Please note that anti-evasion transformations are not applied to this variable by default. REQUEST_BASENAME will recognise both / and \ as path separators.

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 arguments 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 URLENCODED request body processor parsed a request body. This will occur by default when an application/x-www-form-urlencoded is detected, or the URLENCODED request body parser is forced. As of 2.5.7 it is possible to force the presence of the REQUEST_BODY variable, but only when there is

> no request body processor defined, using the `ctl:forceRequestBodyVariable` option in the `REQUEST_HEADERS` phase.

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$" phase:2,t:none,t:normalizePath
```

> **Note**
>
> Please note that anti-evasion transformations are not used on `REQUEST_FILENAME` by default.

REQUEST_HEADERS

This variable can be used as either a collection of all of the request headers or can be used to specify individual 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)$)" t:none,t:lowercase
```

REQUEST_METHOD

This variable holds the request method used by the client.

The following example will trigger if the request method is either `CONNECT` or TRACE.

```
SecRule REQUEST_METHOD "^((?:connect|trace))$" t:none,t:lowercase
```

REQUEST_PROTOCOL

This variable holds the request protocol version information. Example:

```
SecRule REQUEST_PROTOCOL "!^http/(0\.9|1\.0|1\.1)$" t:none,t:lowercase
```

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. It also does not include either the `REQUEST_METHOD` or the HTTP version info.

Example:

```
SecRule REQUEST_URI "attack" phase:1,t:none,t:urlDecode,t:lowercase,t:normalizePath
```

> **Note**
>
> Please note that anti-evasion transformations are not used on `REQUEST_URI` by default.

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).

Example:

```
SecRule REQUEST_URI_RAW "http:/" phase:1,t:none,t:urlDecode,t:lowercase,t:normalizePath
```

> **Note**
>
> Please note that anti-evasion transformations are not used on REQUEST_URI_RAW by default.

RESPONSE_BODY

This variable holds the data for the response payload.

Example:

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

RESPONSE_CONTENT_LENGTH

Response body length in bytes. Can be available starting with phase 3 but it does not have to be (as the length of response body is not always known in advance.) If the size is not known this variable will contain a zero. If RESPONSE_CONTENT_LENGTH contains a zero in phase 5 that means the actual size of the response body was 0.

The value of this variable can change between phases if the body is modified. For example, in embedded mode mod_deflate can compress the response body between phases 4 and 5.

RESPONSE_CONTENT_TYPE

Response content type. Only available starting with phase 3.

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 as 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`, `logdata`, 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 group id (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 user id (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$" \
    "phase:2,t:none,t:lowercase,t:normalizePath,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 request and then it will decided whether or not to allow/deny the request through.

The following is a list of reserved names in the TX collection:

- `TX:0` - The matching value when using the `@rx` or `@pm` operator with the `capture` action.
- `TX:1-TX:9` - The captured subexpression value when using the `@rx` operator with capturing parens and the `capture` action.

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

URLENCODED_ERROR

This flag is raised when an invalid URL encoding is encountered during the parsing of a query string (on every request) or during the parsing of an `application/x-www-form-urlencoded` request body.

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$" skipAfter:12345
SecRule XML:/employees/employee/name/text() Fred
SecRule XML:/xq:employees/employee/name/text() Fred \
    id:12345,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

There are no default transformation functions as there were in previous versions of ModSecurity.

The following rule will ensure that an attacker does not use mixed case in order to evade the ModSecurity rule:

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

multiple transformation 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 ARGS:p "xp_cmdshell" "t:urlDecode,t:lowercase"
```

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

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

The following transformation functions are supported:

base64Decode

This function decodes a base64-encoded string.

base64Encode

This function encodes input string using base64 encoding.

compressWhitespace

It converts whitespace characters (32, \f, \t, \n, \r, \v, 160) to spaces (ASCII 32) and then compresses multiple consecutive space characters into one.

cssDecode

Decodes CSS-encoded characters, as specified at http://www.w3.org/TR/REC-CSS2/syndata.html. This function uses only up to two bytes in the decoding process, meaning it is useful to uncover ASCII characters (that wouldn't normally be encoded) encoded using CSS encoding, or to counter evasion which is a combination of a backslash and non-hexadecimal characters (e.g. `ja\vascript` is equivalent to `javascript`).

escapeSeqDecode

This function decode ANSI C escape sequences: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\\`, `\?`, `\'`, `\"`, `\xHH` (hexadecimal), `\oOOO` (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 `>`

This function will convert any entity into a single byte only, possibly resulting in a loss of information. It is thus useful to uncover bytes that would otherwise not need to be encoded, but it cannot do anything with the characters from the range above 255.

jsDecode

Decodes JavaScript escape sequences. If a `\uHHHH` 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.

length

This function converts the input to its numeric length (count of bytes).

lowercase

This function converts all characters to lowercase using the current C locale.

md5

This function calculates an MD5 hash from input. Note that the computed hash is in a raw binary form and may need encoded into text to be usable (for example: `t:md5,t:hexEncode`).

none

Not an actual transformation function, but an instruction to ModSecurity to remove all transformation functions associated with the current rule.

normalizePath

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

normalizePathWin

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

parityEven7bit

This function calculates even parity of 7-bit data replacing the 8th bit of each target byte with the calculated parity bit.

parityOdd7bit

This function calculates odd parity of 7-bit data replacing the 8th bit of each target byte with the calculated parity bit.

parityZero7bit

This function calculates zero parity of 7-bit data replacing the 8th bit of each target byte with a zero parity bit which allows inspection of even/odd parity 7bit data as ASCII7 data.

removeNulls

This function removes NULL bytes from input.

removeWhitespace

This function removes all whitespace characters from input.

replaceComments

This function replaces each occurrence of a C-style comments (`/* ... */`) with a single space (multiple consecutive occurrences 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 transformation 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. Note that the computed hash is in a raw binary form and may need encoded to be usable (for example: `t:sha1,t:hexEncode`).

trimLeft

This function removes whitespace from the left side of input.

trimRight

This function removes whitespace from the right side of input.

trim

This function removes whitespace from both the left and right sides of input.

Actions

Each action belongs to one of five groups:

Disruptive actions
: Cause ModSecurity to do something. In many cases something means block transaction, but not in all. For example, the allow action is classified as a disruptive action, but it does the opposite of blocking. There can only be one disruptive action per rule (if there are multiple disruptive actions present, or inherited, only the last one will

take effect), or rule chain (in a chain, a disruptive action can only appear in the first rule).

Non-disruptive actions
: Do something, but that something does not and cannot affect the rule processing flow. Setting a variable, or changing its value is an example of a non-disruptive action. Non-disruptive action can appear in any rule, including each rule belonging to a chain.

Flow actions
: These actions affect the rule flow (for example `skip` or `skipAfter`).

Meta-data actions
: Meta-data actions are used to provide more information about rules. Examples include `id`, `rev`, `severity` and `msg`.

Data actions
: Not really actions, these are mere containers that hold data used by other actions. For example, the `status` action holds the status that will be used for blocking (if it takes place).

`allow`

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

Action Group: Disruptive

Example:

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

Prior to ModSecurity 2.5 the `allow` action would only affect the current phase. An `allow` in phase 1 would skip processing the remaining rules in phase 1 but the rules from phase 2 would execute. Starting with v2.5.0 `allow` was enhanced to allow for fine-grained control of what is done. The following rules now apply:

1. If used one its own, like in the example above, `allow` will affect the entire transaction, stopping processing of the current phase but also skipping over all other phases apart from the logging phase. (The logging phase is special; it is designed to always execute.)
2. If used with parameter "phase", `allow` will cause the engine to stop processing the current phase. Other phases will continue as normal.
3. If used with parameter "request", `allow` will cause the engine to stop processing the current phase. The next phase to be processed will be phase `RESPONSE_HEADERS`.

Examples:

```
# Do not process request but process response.
SecAction phase:1,allow:request

# Do not process transaction (request and response).
SecAction phase:1,allow
```

If you want to allow a response through, put a rule in phase `RESPONSE_HEADERS` and simply use `allow` on its own:

```
# Allow response through.
SecAction phase:3,allow
```

append

Description: Appends text given as parameter to the end of response body. For this action to work content injection must be enabled by setting `SecContentInjection` to `On`. Also make sure you check the content type of the response before you make changes to it (e.g. you don't want to inject stuff into images).

Action Group: Non-disruptive

Processing Phases: 3 and 4.

Example:

```
SecRule RESPONSE_CONTENT_TYPE "^text/html" "nolog,pass,append:'<hr>Footer'"
```

> **Note**
>
> While macro expansion is allowed in the additional content, you are strongly cautioned against inserting user defined data fields.

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.

block

Description: Performs the default disruptive action.

Action Group: Disruptive

It is intended to be used by ruleset writers to signify that the rule was intended to block and leaves the "how" up to the administrator. This action is currently a placeholder which will just be replaced by the action from the last `SecDefaultAction` in the same context. Using the `block` action with the `SecRuleUpdateActionById` directive allows a rule to be reverted back to the previous `SecDefaultAction` disruptive action.

In future versions of ModSecurity, more control and functionality will be added to define "how" to block.

Examples:

In the following example, the second rule will "deny" because of the SecDefaultAction disruptive action. The intent being that the administrator could easily change this to another disruptive action without editing the actual rules.

```
### Administrator defines "how" to block (deny,status:403)...
SecDefaultAction phase:2,deny,status:403,log,auditlog

### Included from a rulest...
# Intent is to warn for this User Agent
SecRule REQUEST_HEADERS:User-Agent "perl" "phase:2,pass,msg:'Perl based user agent identified'"
# Intent is to block for this User Agent, "how" described in SecDefaultAction
SecRule REQUEST_HEADERS:User-Agent "nikto" "phase:2,block,msg:'Nikto Scanners Identified'"
```

In the following example, The rule is reverted back to the `pass` action defined in the SecDefaultAction directive by using the `SecRuleUpdateActionById` directive in conjuction with the `block` action. This allows an administrator to override an action in a 3rd party rule without modifying the rule itself.

```
### Administrator defines "how" to block (deny,status:403)...
SecDefaultAction phase:2,pass,log,auditlog

### Included from a rulest...
SecRule REQUEST_HEADERS:User-Agent "nikto" "id:1,phase:2,deny,msg:'Nikto Scanners Identified'"

### Added by the administrator
SecRuleUpdateActionById 1 "block"
```

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 parens, 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. Do note that
# this rule should be preceeded by a rule that verifies
# only valid request methods (e.g. GET, HEAD and POST) are used.
SecRule REQUEST_METHOD ^POST$ chain,t:none
SecRule REQUEST_HEADERS:Content-Length ^$ t:none
```

> **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), skip and skipAfter 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. ruleRemoveById (single rule ID, or a single rule ID range accepted as parameter)
5. requestBodyAccess
6. forceRequestBodyVariable
7. requestBodyLimit
8. requestBodyProcessor
9. responseBodyAccess
10. responseBodyLimit
11. ruleEngine

With the exception of requestBodyProcessor and forceRequestBodyVariable, 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 bodies, 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.

The `forceRequestBodyVariable` option allows you to configure the `REQUEST_BODY` variable to be set when there is no request body processor configured. This allows for inspection of request bodies of unknown types.

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 phase:1,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 currently not available on Windows based builds. 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. As of v2.5.0, if the parameter supplied to exec is a Lua script (detected by the `.lua` extension) the script will be processed *internally*. This means you will get direct access to the internal request context from the script. Please read the `SecRuleScript` documentation for more details on how to write Lua scripts.

Action Group: Non-disruptive

Example:

```
# The following is going to execute /usr/local/apache/bin/test.sh
# as a shell script on rule match.
SecRule REQUEST_URI "^/cgi-bin/script\.pl" \
    "phase:2,t:none,t:lowercase,t:normalizePath,log,exec:/usr/local/apache/bin/test.sh"

# The following is going to process /usr/local/apache/conf/exec.lua
# internally as a Lua script on rule match.
SecRule ARGS:p attack log,exec:/usr/local/apache/conf/exec.lua
```

> **Note**
>
> The exec action is executed independently from any disruptive actions. External scripts will always be called with no parameters. Some transaction information will be placed in environment variables. All the usual CGI environment variables will be there. 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 multi-threaded operation. The script you execute must write something (anything) to stdout. If it doesn't ModSecurity will assume execution didn't work.

expirevar

Description: Configures a collection 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" \
    "phase:2,t:none,t:lowercase,t:normalizePath,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 indented 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: Meta-data

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 phase:1,initcol:ip=%{REMOTE_ADDR},nolog
```

Note

Normally you will want to use *phase:1* along with *initcol* so that the collection is available in all phases.

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

See the "Persistant Storage" section for further details.

log

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

Action Group: Non-disruptive

Example:

```
SecAction phase:1,initcol:ip=%{REMOTE_ADDR},log
```

Note

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

logdata

Description: Allows a data fragment to be logged as part of the alert message.

Action Group: Non-disruptive

Example:

```
SecRule &ARGS:p "@eq 0" "log,logdata:'%{TX.0}'"
```

Note

The logdata information appears in the error and/or audit log files and is not sent back to the client in response headers. Macro expansion is preformed so you may use variable names such as %{TX.0}, etc. The information is properly escaped for use with logging binary data.

msg

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

Action Group: Meta-data

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 response 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 transaction 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

Example1:

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

When using *pass* with SecRule with multiple targets, *all* targets will be processed and *all* non-disruptive actions will trigger for *every* match found. In the second example the TX:test target would be incremented by 1 for each matching argument.

Example2:

```
SecRule ARGS "test" log,pass,setvar:TX.test=+1
```

Note

The transaction will not be interrupted but a log will be generated for each matching target (unless logging has been suppressed).

pause

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

Action Group: Non-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: Meta-data

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.

prepend

Description: Prepends text given as parameter to the response body. For this action to work content injection must be enabled by setting `SecContentInjection` to `On`. Also make sure you check the content type of the response before you make changes to it (e.g. you don't want to inject stuff into images).

Action Group: Non-disruptive

Processing Phases: 3 and 4.

Example:

```
SecRule RESPONSE_CONTENT_TYPE ^text/html "phase:3,nolog,pass,prepend:'Header<br>'"
```

> **Note**
>
> While macro expansion is allowed in the additional content, you are strongly cautioned against inserting user defined data fields.

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: Meta-data

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

Assigns severity to the rule it is used with.

Action Group: Meta-data

Example:

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

Note

Severity values in ModSecurity follow those of syslog, as shown in below:

Table 14.1. Severity values

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

It is possible to specify severity levels using either the numerical values or the text values, but you should always specify severity levels using the text values because it is difficult to

remember what a number stands for. The use of the numerical values is deprecated as of v2.5.0 and may be removed in one of the susequent major updates.

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-purpose action 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 predefined 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: Flow

Example:

```
SecRule REQUEST_URI "^/$" \
"phase:2,chain,t:none,skip:2"
SecRule REMOTE_ADDR "^127\.0\.0\.1$" "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 parameter denoting the number of rules (or chains) to skip.

skipAfter

Description: Skips rules (or chains) on successful match resuming rule execution after the specified rule ID or marker (see `SecMarker`) is found.

Action Group: Flow

Example:

```
SecRule REQUEST_URI "^/$" "chain,t:none,skipAfter:960015"
SecRule REMOTE_ADDR "^127\.0\.0\.1$" "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

`SkipAfter` 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 parameter denoting the last rule ID to skip.

status

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

Action Group: Data

Example:

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

Note

Status actions defined in Apache scope locations (such as Directory, Location, etc...) may be superseded 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,t:hexEncode
```

Note

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

tag

Description: Assigns custom text to a rule or chain.

Action Group: Meta-data

Example:

```
SecRule REQUEST_FILENAME "\b(?:n(?:map|et|c)|w(?:guest|sh)|cmd(?:32)?|telnet|rcmd|ftp)\.exe\b" \
    "t:none,t:lowercase,deny,msg:'System Command Access',id:'950002',\
tag:'WEB_ATTACK/FILE_INJECTION',tag:'OWASP/A2',severity:'2'"
```

Note

The tag information appears in the error and/or audit log files. Its intent is to be used to automate classification of rules and the alerts generated by rules. Multiple tags can be used per rule/chain.

xmlns

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

Action Group: Data

Example:

```
SecRule REQUEST_HEADERS:Content-Type "text/xml" \
```

```
    "phase:1,pass,ctl:requestBodyProcessor=XML,ctl:requestBodyAccess=On, \
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
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 uses the @ symbol followed by the specific operator name.

beginsWith

Description: This operator is a string comparison and returns true if the parameter value is found at the beginning of the input. Macro expansion is performed so you may use variable names such as %{TX.1}, etc.

Example:

```
SecRule REQUEST_LINE "!@beginsWith GET" t:none,deny,status:403
SecRule REQUEST_ADDR "^(.*)\.\d+$" deny,status:403,capture,chain
SecRule ARGS:gw "!@beginsWith %{TX.1}"
```

contains

Description: This operator is a string comparison and returns true if the parameter value is found anywhere in the input. Macro expansion is performed so you may use variable names such as %{TX.1}, etc.

Example:

```
SecRule REQUEST_LINE "!@contains .php" t:none,deny,status:403
SecRule REQUEST_ADDR "^(.*)$" deny,status:403,capture,chain
SecRule ARGS:ip "!@contains %{TX.1}"
```

endsWith

Description: This operator is a string comparison and returns true if the parameter value is found at the end of the input. Macro expansion is performed so you may use variable names such as %{TX.1}, etc.

Example:

```
SecRule REQUEST_LINE "!@endsWith HTTP/1.1" t:none,deny,status:403
SecRule ARGS:route "!@endsWith %{REQUEST_ADDR}" t:none,deny,status:403
```

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"
```

geoLookup

Description: This operator looks up various data fields from an IP address or hostname in the target data. The results will be captured in the `GEO` collection.

You must provide a database via `SecGeoLookupDb` before this operator can be used.

> **Note**
>
> This operator matches and the action is executed on a *successful* lookup. For this reason, you probably want to use the *pass,nolog* actions. This allows for `setvar` and other non-disruptive actions to be executed on a match. If you wish to block on a failed lookup, then do something like this (look for an empty GEO collection):

```
SecGeoLookupDb /usr/local/geo/data/GeoLiteCity.dat
...
SecRule REMOTE_ADDR "@geoLookup" "pass,nolog"
SecRule &GEO "@eq 0" "deny,status:403,msg:'Failed to lookup IP'"
```

See the `GEO` variable for an example and more information on various fields available.

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. As of v2.5.0, if the supplied filename is not absolute it is treated as relative to the directory in which the configuration file resides. Also as of v2.5.0, if the filename is determined to be a Lua script (based on its extension) the script will be processed by the internal engine. As such it will have full access to the ModSecurity context.

Example of using an external binary/script:

```
# Execute external script to validate uploaded files.
SecRule FILES_TMPNAMES "@inspectFile /opt/apache/bin/inspect_script.pl"
```

Example of using Lua script:

```
SecRule FILES_TMPNANMES "@inspectFile inspect.lua"
```

Script `inspect.lua`:

```
function main(filename)
    -- Do something to the file to verify it. In this example, we
    -- read up to 10 characters from the beginning of the file.
    local f = io.open(filename, "rb");
    local d = f:read(10);
    f:close();

    -- Return null if there is no reason to believe there is ansything
    -- wrong with the file (no match). Returning any text will be taken
    -- to mean a match should be trigerred.
    return null;
end
```

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"
```

pm

Description: Phrase Match operator. This operator uses a set based matching engine (Aho-Corasick) for faster matches of keyword lists. It will match any one of its arguments anywhere in the target value. The match is case insensitive.

Example:

```
SecRule REQUEST_HEADERS:User-Agent "@pm WebZIP WebCopier Webster WebStripper SiteSnagger ProWebWalker
```

The above would deny access with 403 if any of the words matched within the User-Agent HTTP header value.

pmFromFile

Description: Phrase Match operator. This operator uses a set based matching engine (Aho-Corasick) for faster matches of keyword lists. This operator is the same as `@pm` except that it takes a list of files as arguments. It will match any one of the phrases listed in the file(s) anywhere in the target value.

Notes:

1. The contents of the files should be one phrase per line. End of line markers will be stripped from the phrases, however, whitespace will not be trimmed from phrases in the file. Empty lines and comment lines (beginning with a '#') are ignored.
2. To allow easier inclusion of phrase files with rulesets, relative paths may be used to the phrase files. In this case, the path of the file containing the rule is prepended to the phrase file path.

Example:

```
SecRule REQUEST_HEADERS:User-Agent "@pm /path/to/blacklist1 blacklist2" "deny,status:403
```

The above would deny access with 403 if any of the patterns in the two files matched within the User-Agent HTTP header value. The `blacklist2` file would need to be placed in the same path as the file containing the rule.

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` transformation function, or use the per-pattern `(?i)` modifier, 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 character.

streq

Description: This operator is a string comparison and returns true if the parameter value matches the input exactly. Macro expansion is performed so you may use variable names such as %{TX.1}, etc.

Example:

```
SecRule ARGS:foo "!@streq bar" t:none,deny,status:403
SecRule REQUEST_ADDR "^(.*)$" deny,status:403,capture,chain
SecRule REQUEST_HEADERS:Ip-Address "!@streq %{TX.1}"
```

validateByteRange

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

Example:

```
SecRule ARGS: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: Validates the DOM tree generated by the XML request body processor against the supplied DTD.

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,skipAfter:12345
SecRule XML "@validateDTD /path/to/apache2/conf/xml.dtd" "deny,id:12345"
```

> **Note**
>
> This operator requires request body to be processed as XML.

validateSchema

Description: Validates the DOM tree generated by the XML request body processor against the supplied XML Schema.

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,skipAfter:12345
SecRule XML "@validateSchema /path/to/apache2/conf/xml.xsd" "deny,id:12345"
```

> **Note**
>
> 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).

verifyCC

Description: This operator verifies a given regular expression as a potential credit card number. It first matches with a single generic regular expression then runs the resulting match through a Luhn checksum algorithm to further verify it as a potential credit card number.

Example:

```
SecRule ARGS "@verifyCC \d{13,16}" \
             "phase:2,sanitiseMatched,log,auditlog,pass,msg:'Potential credit card number'"
```

within

Description: This operator is a string comparison and returns true if the input value is found anywhere within the parameter value. Note that this is similar to `@contains`, except that the target and match values are reversed. Macro expansion is performed so you may use variable names such as %{TX.1}, etc.

Example:

```
SecRule REQUEST_METHOD "!@within get,post,head" t:lowercase,deny,status:403

SecAction "pass,setvar:'tx.allowed_methods=get,post,head'"
SecRule REQUEST_METHOD "!@within %{tx.allowed_methods}" t:lowercase,deny,status:403
```

15 Data Formats Guide

The purpose of this document is to describe the formats of the ModSecurity alert messages, transaction logs and communication protocols, which would not only allow for a better understanding what ModSecurity does but also for an easy integration with third-party tools and products.

Alerts

As part of its operations ModSecurity will emit alerts, which are either *warnings* (non-fatal) or *errors* (fatal, usually leading to the interception of the transaction in question). Below is an example of a ModSecurity alert entry:

```
Access denied with code 505 (phase 1). Match of "rx …
^HTTP/(0\\\\.9|1\\\\.[01])$" against "REQUEST_PROTOCOL" required. …
[id "960034"] [msg "HTTP protocol version is not allowed by policy"] …
[severity "CRITICAL"] [uri "/"] [unique_id "PQaTTVBEUOkAAFwKXrYAAAAM"]
```

> **Note**
>
> Alerts will only ever contain one line of text but we've broken the above example into multiple lines to make it fit into the page.

Each alert entry begins with the engine message, which describes what ModSecurity did and why. For example:

```
Access denied with code 505 (phase 1). Match of "rx …
^HTTP/(0\\\\.9|1\\\\.[01])$" against "REQUEST_PROTOCOL" required.
```

Alert Action Description

The first part of the engine message tells you whether ModSecurity acted to interrupt transaction or rule processing:

1. If the alert is only a warning, the first sentence will simply say *Warning*.
2. If the transaction was intercepted, the first sentence will begin with *Access denied*. What follows is the list of possible messages related to transaction interception:
   - *Access denied with code %0* - a response with status code `%0` was sent.
   - *Access denied with connection close* - connection was abruptly closed.
   - *Access denied with redirection to %0 using status %1* - a redirection to URI `%0` was issued using status `%1`.
3. There is also a special message that ModSecurity emits where an `allow` action is executed. There are three variations of this type of message:
   - *Access allowed* - rule engine stopped processing rules (transaction was unaffected).
   - *Access to phase allowed* - rule engine stopped processing rules in the current phase only. Subsequent phases will be processed normally. Transaction was not affected by this rule but it may be affected by any of the rules in the subsequent phase.
   - *Access to request allowed* - rule engine stopped processing rules in the current phase. Phases prior to request execution in the backend (currently phases 1 and 2) will not be processed. The response phases (currently phases 3 and 4) and others (currently phase 5) will be processed as normal. Transaction was not affected by this rule but it may be affected by any of the rules in the subsequent phase.

Alert Justification Description

The second part of the engine message explains *why* the alert was generated. Since it is automatically generated from the rules it will be very technical in nature, talking about operators and their parameters and give you insight into what the rule looked like. But this message cannot give you insight into the reasoning behind the rule. A well-written rule will always specify a human-readable message (using the `msg` action) to provide further information.

The format of the second part of the engine message depends on whether it was generated by the operator (which happens on a match) or by the rule processor (which happens where there is not a match, but the negation was used):

- `@beginsWith` - *String match %0 at %1.*
- `@contains` - *String match %0 at %1.*
- `@containsWord` - *String match %0 at %1.*
- `@endsWith` - *String match %0 at %1.*
- `@eq` - *Operator EQ matched %0 at %1.*
- `@ge` - *Operator GE matched %0 at %1.*
- `@geoLookup` - *Geo lookup for %0 succeeded at %1.*

- `@inspectFile` - *File %0 rejected by the approver script %1: %2*
- `@le` - *Operator LE matched %0 at %1.*
- `@lt` - *Operator LT matched %0 at %1.*
- `@rbl` - *RBL lookup of %0 succeeded at %1.*
- `@rx` - *Pattern match %0 at %1.*
- `@streq` - *String match %0 at %1.*
- `@validateByteRange` - *Found %0 byte(s) in %1 outside range: %2.*
- `@validateDTD` - *XML: DTD validation failed.*
- `@validateSchema` - *XML: Schema validation failed.*
- `@validateUrlEncoding`
  - *Invalid URL Encoding: Non-hexadecimal digits used at %0.*
  - *Invalid URL Encoding: Not enough characters at the end of input at %0.*
- `@validateUtf8Encoding`
  - *Invalid UTF-8 encoding: not enough bytes in character at %0.*
  - *Invalid UTF-8 encoding: invalid byte value in character at %0.*
  - *Invalid UTF-8 encoding: overlong character detected at %0.*
  - *Invalid UTF-8 encoding: use of restricted character at %0.*
  - *Invalid UTF-8 encoding: decoding error at %0.*
- `@verifyCC` - *CC# match %0 at %1.*

Messages not related to operators:

- When `SecAction` directive is processed - *Unconditional match in SecAction.*
- When `SecRule` does not match but negation is used - *Match of %0 against %1 required.*

> **Note**
>
> The parameters to the operators `@rx` and `@pm` (regular expression and text pattern, respectively) will be truncated to 252 bytes if they are longer than this limit. In this case the parameter in the alert message will be terminated with three dots.

Meta-data

The metadata fields are always placed at the end of the alert entry. Each metadata field is a text fragment that consists of an open bracket followed by the metadata field name, followed by the value and the closing bracket. What follows is the text fragment that makes up the `id` metadata field.

```
[id "960034"]
```

The following metadata fields are currently used:

1. `offset` - The byte offset where a match occured within the target data. This is not always available.
2. `id` - Unique rule ID, as specified by the `id` action.
3. `rev` - Rule revision, as specified by the `rev` action.
4. `msg` - Human-readable message, as specified by the `msg` action.
5. `severity` - Event severity as text, as specified by the `severity` action. The possible values (with their corresponding numberical values in brackets) are `EMERGENCY` (0), `ALERT` (1), `CRITICAL` (2), `ERROR` (3), `WARNING` (4), `NOTICE` (5), `INFO` (6) and `DEBUG` (7).
6. `unique_id` - Unique event ID, generated automatically.
7. `uri` - Request URI.
8. `logdata` - contains transaction data fragment, as specified by the `logdata` action.

Escaping

ModSecurity alerts will always contain text fragments that were taken from configuration or the transaction. Such text fragments escaped before they are user in messages, in order to sanitise the potentially dangerous characters. They are also sometimes surrounded using double quotes. The escaping algorithm is as follows:

1. Characters `0x08` (`BACKSPACE`), `0x0a` (`NEWLINE`), `0x10` (`CARRIAGE RETURN`), `0x09` (`HORIZONTAL TAB`) and `0x0b` (`VERTICAL TAB`) will be represented as `\b`, `\n`, `\r`, `\t` and `\v`, respectively.
2. Bytes from the ranges `0-0x1f` and `0x7f-0xff` (inclusive) will be represented as `\xHH`, where `HH` is the hexadecimal value of the byte.
3. Backslash characters (`\`) will be represented as `\\`.
4. Each double quote character will be represented as `\"`, but only if the entire fragment is surrounded with double quotes.

Alerts in the Apache Error Log

Every ModSecurity alert conforms to the following format when it appears in the Apache error log:

```
[Sun Jun 24 10:19:58 2007] [error] [client 192.168.0.1] …
ModSecurity: ALERT_MESSAGE
```

The above is a standard Apache error log format. The `ModSecurity:` prefix is specific to ModSecurity. It is used to allow quick identification of ModSecurity alert messages when they appear in the same file next to other Apache messages.

The actual message (`ALERT_MESSAGE` in the example above) is in the same format as described in the *Alerts* section.

> **Note**
>
> Apache further escapes ModSecurity alert messages before writing them to the error log. This means that all backslash characters will be doubled in the error log. In practice, since ModSecurity will already represent a single backslash within an untrusted text fragment as two backslashes, the end result in the Apache error log will be *four* backslashes. Thus, if you need to interpret a ModSecurity message from the error log, you should decode the message part after the `ModSecurity:` prefix first. This step will peel the first encoding layer.

Alerts in Audit Logs

Alerts are transported in the `H` section of the ModSecurity Audit Log. Alerts will appear each on a separate line and in the order they were generated by ModSecurity. Each line will be in the following format:

```
Message: ALERT_MESSAGE
```

Below is an example of an `H` section that contains two alert messages:

```
--c7036611-H--
Message: Warning. Match of "rx ^apache.*perl" against …
"REQUEST_HEADERS:User-Agent" required. [id "990011"] [msg "Request …
Indicates an automated program explored the site"] [severity "NOTICE"]
Message: Warning. Pattern match "(?:\\b(?:(?:s(?:elect\\b(?:.{1,100}?\\b…
(?:(?:length|count|top)\\b.{1,100}?\\bfrom|from\\b.{1,100}?\\bwhere)…
|.*?\\b(?:d(?:ump\\b.*\\bfrom|ata_type)|(?:to_(?:numbe|cha)|inst)r))|p_…
(?:(?:addextendedpro|sqlexe)c|(?:oacreat|prepar)e|execute(?:sql)?|…
makewebt ..." at ARGS:c. [id "950001"] [msg "SQL Injection Attack. …
Matched signature: union select"] [severity "CRITICAL"]
Stopwatch: 1199881676978327 2514 (396 2224 -)
Producer: ModSecurity v2.x.x (Apache 2.x)
Server: Apache/2.x.x

--c7036611-Z--
```

Audit Log

ModSecurity records one transaction in a single audit log file. Below is an example:

```
--c7036611-A--
[09/Jan/2008:12:27:56 +0000] OSD4l1BEUOkAAHZ8Y3QAAAAH 209.90.77.54 64995
  80.68.80.233 80
--c7036611-B--
GET //EvilBoard_0.1a/index.php?c='/**/union/**/select/**/1,concat(username,…
char(77),password,char(77),email_address,char(77),info,char(77),user_level,…
char(77))/**/from/**/eb_members/**/where/**/userid=1/*http://kamloopstutor.…
com/images/banners/on.txt? HTTP/1.1
TE: deflate,gzip;q=0.3
Connection: TE, cslose
Host: www.example.com
User-Agent: libwww-perl/5.808

--c7036611-F--
HTTP/1.1 404 Not Found
Content-Length: 223
Connection: close
Content-Type: text/html; charset=iso-8859-1

--c7036611-H--
Message: Warning. Match of "rx ^apache.*perl" against …
"REQUEST_HEADERS:User-Agent" required. [id "990011"] [msg "Request …
Indicates an automated program explored the site"] [severity "NOTICE"]
Message: Warning. Pattern match "(?:\\b(?:(?:s(?:elect\\b(?:.{1,100}?\\b…
(?:(?:length|count|top)\\b.{1,100}?\\bfrom|from\\b.{1,100}?\\bwhere)…
|.*?\\b(?:d(?:ump\\b.*\\bfrom|ata_type)|(?:to_(?:numbe|cha)|inst)r))|p_…
(?:(?:addextendedpro|sqlexe)c|(?:oacreat|prepar)e|execute(?:sql)?|…
makewebt ..." at ARGS:c. [id "950001"] [msg "SQL Injection Attack. …
Matched signature: union select"] [severity "CRITICAL"]
Stopwatch: 1199881676978327 2514 (396 2224 -)
Producer: ModSecurity v2.x.x (Apache 2.x)
Server: Apache/2.x.x

--c7036611-Z--
```

The file consist of multiple sections, each in different format. Separators are used to define sections:

```
--c7036611-A--
```

A separator always begins on a new line and conforms to the following format:

1. Two dashes

2. Unique boundary, which consists from several hexadecimal characters.
3. One dash character.
4. Section identifier, currently a single uppercase letter.
5. Two trailing dashes.

Refer to the documentation for `SecAuditLogParts` for the explanation of each part.

Parts

This section documents the audit log parts available in ModSecurity 2.x. They are:

- `A` - audit log header
- `B` - request headers
- `C` - request body
- `D` - intended response headers (NOT IMPLEMENTED)
- `E` - intended response body
- `F` - response headers
- `G` - response body (NOT IMPLEMENTED)
- `H` - audit log trailer
- `I` - reduced multipart request body
- `J` - multipart files information (NOT IMPLEMENTED)
- `K` - matched rules information
- `Z` - audit log footer

Audit Log Header (A)

ModSecurity 2.x audit log entries always begin with the header part. For example:

```
--c7036611-A--
[09/Jan/2008:12:27:56 +0000] OSD4l1BEU0kAAHZ8Y3QAAAAH 209.90.77.54 64995 …
80.68.80.233 80
```

The header contains only one line, with the following information on it:

1. Timestamp
2. Unique transaction ID
3. Source IP address (IPv4 or IPv6)
4. Source port
5. Destination IP address (IPv4 or IPv6)

6. Destination port

Request Headers (B)

The request headers part contains the request line and the request headers. The information present in this part will not be identical to that sent by the client responsible for the transaction. ModSecurity 2.x for Apache does not have access to the raw data; it sees what Apache itself sees. While the end result may be identical to the raw request, differences are possible in some areas:

1. If any of the fields are `NUL`-terminated, Apache will only see the content prior to the NUL.
2. Headers that span multiple lines (feature known as header folding) will be collapsed into a single line.
3. Multiple headers with the same name will be combined into a single header (as allowed by the HTTP RFC).

Request Body (C)

This part contains the request body of the transaction, after dechunking and decompression (if applicable).

Intended Response Headers (D)

This part contains the status line and the request headers that would have been delivered to the client had ModSecurity not intervened. Thus this part makes sense only for transactions where ModSecurity altered the data flow. By differentiating before the intended and the final response headers, we are able to record what was internally ready for sending, but also what was actually sent.

> **Note**
>
> This part is reserved for future use. It is not currently implemented in ModSecurity 2.x.

Intended Response Body (E)

This part contains the transaction response body (before compression and chunking, where used) that was either sent or would have been sent had ModSecurity not intervened. You can find whether interception took place by looking at the `Action` header of the part `H`. If that

header is present, and the interception took place in phase 3 or 4 then the E part contains the intended response body. Otherwise, it contains the actual response body.

> **Note**
>
> Once the G (actual response body) part is implemented, part E will be present only in audit logs that contain a transaction that was intercepted, and there will be no need for further analsys.

Response Headers (F)

This part contains the actual response headers sent to the client. Since ModSecurity 2.x for Apache does not access the raw connection data, it constructs part F out of the internal Apache data structures that hold the response headers.

Some headers (the Date and Server response headers) are generated just before they are sent and ModSecurity is not able to record those. You should note than ModSecurity is working as part of a reverse proxy, the backend web server will have generated these two servers, and in that case they will be recorded.

Response Body (G)

When implemented, this part will contain the actual response body before compression and chunking.

> **Note**
>
> This part is reserved for future use. It is not implemented in ModSecurity 2.x.

Audit Log Trailer (H)

Part H contains additional transaction meta-data that was obtained from the web server or from ModSecurity itself. The part contains a number of trailer headers, which are similar to HTTP headers (without support for header folding):

1. Action
2. Apache-Error
3. Message
4. Producer
5. Response-Body-Transformed
6. Sanitised-Args

7. Sanitised-Request-Headers
8. Sanitised-Response-Headers
9. Server
10. Stopwatch
11. WebApp-Info

Action

The `Action` header is present only for the transactions that were intercepted:

```
Action: Intercepted (phase 2)
```

The phase information documents the phase in which the decision to intercept took place.

Apache-Error

The Apache-Error header contains Apache error log messages observed by ModSecurity, excluding those sent by ModSecurity itself. For example:

```
Apache-Error: [file "/tmp/buildd/apache2-2.0.54/build-tree/apache2/server/…
core.c"] [line 3505] [level 3] File does not exist: /var/www/www.…
modsecurity.org/fst/documentation/modsecurity-apache/2.5.0-dev2
```

Message

Zero or more `Message` headers can be present in any trailer, and each such header will represent a single ModSecurity warning or error, displayed in the order they were raised.

The example below was broken into multiple lines to make it fit this page:

```
Message: Access denied with code 400 (phase 2). Pattern match "^\w+:/" at …
REQUEST_URI_RAW. [file "/etc/apache2/rules-1.6.1/modsecurity_crs_20_…
protocol_violations.conf"] [line "74"] [id "960014"] [msg "Proxy access …
attempt"] [severity "CRITICAL"] [tag "PROTOCOL_VIOLATION/PROXY_ACCESS"]
```

Producer

The `Producer` header identifies the product that generated the audit log. For example:

```
Producer: ModSecurity for Apache/2.5.5 (http://www.modsecurity.org/).
```

ModSecurity allows rule sets to add their own signatures to the `Producer` information (this is done using the `SecComponentSignature` directive). Below is an example of the `Producer` header with the signature of one component (all one line):

```
Producer: ModSecurity for Apache/2.5.5 (http://www.modsecurity.org/);…
MyComponent/1.0.0 (Beta).
```

Response-Body-Transformed

This header will appear in every audit log that contains a response body:

```
Response-Body-Transformed: Dechunked
```

The contents of the header is constant at present, so the header is only useful as a reminder that the recorded response body is not identical to the one sent to the client. The actual content is the same, except that Apache may further compress the body and deliver it in chunks.

Sanitised-Args

The `Sanitised-Args` header contains a list of arguments that were sanitised (each byte of their content replaced with an asterisk) before logging. For example:

```
Sanitised-Args: "old_password", "new_password", "new_password_repeat".
```

Sanitised-Request-Headers

The `Sanitised-Request-Headers` header contains a list of request headers that were sanitised before logging. For example:

```
Sanitised-Request-Headers: "Authentication".
```

Sanitised-Response-Headers

The `Sanitised-Response-Headers` header contains a list of response headers that were sanitised before logging. For example:

```
Sanitised-Response-Headers: "My-Custom-Header".
```

Server

The `Server` header identifies the web server. For example:

```
Server: Apache/2.0.54 (Debian GNU/Linux) mod_ssl/2.0.54 OpenSSL/0.9.7e
```

This information may sometimes be present in any of the parts that contain response headers, but there are a few cases when it isn't:

1. None of the response headers were recoreded.

2. The information in the response headers is not accurate because server signature masking was used.

Stopwatch

The `Stopwatch` header provides certain diagnostic information that allows you to determine the performance of the web server and of ModSecurity itself. It will typically look like this:

```
Stopwatch: 1222945098201902 2118976 (770* 4400 -)
```

Each line can contain up to 5 different values. Some values can be absent; each absent value will be replaced with a dash.

The meanings of the values are as follows (all values are in microseconds):

1. Transaction timestamp in microseconds since January 1st, 1970.
2. Transaction duration.
3. The time between the moment Apache started processing the request and until phase 2 of ModSecurity began. If an asterisk is present that means the time includes the time it took ModSecurity to read the request body from the client (typically slow). This value can be used to provide a rough estimate of the client speed, but only with larger request bodies (the smaller request bodies may arrive in a single TCP/IP packet).
4. The time between the start of processing and until phase 2 was completed. If you substract the previous value from this value you will get the exact duration of phase 2 (which is the main rule processing phase).
5. The time between the start of request processing and util we began sending a fully-buffered response body to the client. If you substract this value from the total transaction duration and divide with the response body size you may get a rough estimate of the client speed, but only for larger response bodies.

WebApp-Info

The `WebApp-Info` header contains information on the application to which the recorded transaction belongs. This information will appear only if it is known, which will happen if `SecWebAppId` was set, or `setsid` or `setuid` executed in the transaction.

The header uses the following format:

```
WebApp-Info: "WEBAPPID" "SESSIONID" "USERID"
```

Each unknown value is replaced with a dash.

Reduced Multipart Request Body (I)

Transactions that deal with file uploads tend to be large, yet the file contents is not always relevant from the security point of view. The `I` part was designed to avoid recording raw `multipart/form-data` request bodies, replacing them with a simulated `application/x-www-form-urlencoded` body that contains the same key-value parameters.

The reduced multipart request body will not contain any file information. The `J` part (currently not implemented) is intended to carry the file metadata.

Multipart Files Information (J)

The purpose of part `J` is to record the information on the files contained in a `multipart/form-data` request body. This is handy in the cases when the original request body was not recorded, or when only a reduced version was recorded (e.g. when part `I` was used instead of part `C`).

> **Note**
>
> This part is reserved for future use. It is not implemented in ModSecurity 2.x.

Matched Rules (K)

The matched rules part contains a record of all ModSecurity rules that matched during transaction processing. You should note that if a rule that belongs to a chain matches then the entire chain will be recorded. This is because, even though the disruptive action may not have executed, other per-rule actions have, and you will need to see the entire chain in order to understand the rules.

This part is available starting with ModSecurity 2.5.x.

Audit Log Footer (Z)

Part `Z` is a special part that only has a boundary but no content. Its only purpose is to signal the end of an audit log.

Storage Formats

ModSecurity supports two audit log storage formats:

1. *Serial* audit log format - multiple audit log files stored in the same file.

2. *Concurrent* audit log format - one file is used for every audit log.

Serial Audit Log Format

The serial audit log format stores multiple audit log entries within the same file (one after another). This is often very convinent (audit log entries are easy to find) but this format is only suitable for light logging in the current ModSecurity implementation because writing to the file is serialised: only one audit log entry can be written at any one time.

Concurrent Audit Log Format

The concurrent audit log format uses one file per audit log entry, and allows many transactions to be recorded at once. A hierarchical directory structure is used to ensure that the number of files created in any one directory remains relatively small. For example:

```
$LOGGING-HOME/20081128/20081128-1414/20081128-141417-…
egDKy38AAAEAAAyMHXsAAAAA
```

The current time is used to work out the directory structure. The file name is constructed using the current time and the transaction ID.

The creation of every audit log in concurrent format is recorded with an entry in the concurrent audit log *index file*. The format of each line resembles the common web server access log format. For example:

```
192.168.0.111 192.168.0.1 - - [28/Nov/2008:15:06:32 +0000] …
"GET /?p=\\ HTTP/1.1" 200 69 "-" "-" NOfRx38AAAEAAAzcCU4AAAAA …
"-" /20081128/20081128-1506/20081128-150632-NOfRx38AAAEAAAzcCU4AAAAA …
0 1183 md5:ffee2d414cd43c2f8ae151652910ed96
```

The tokens on the line are as follows:

1. Hostname (or IP address, if the hostname is not known)
2. Source IP address
3. Remote user (from HTTP Authentication)
4. Local user (from identd)
5. Timestamp
6. Request line
7. Response status
8. Bytes sent (in the response body)
9. Referrer information
10. User-Agent information

11. Transaction ID
12. Session ID
13. Audit log file name (relative to the audit logging home, as configured using the `SecAuditLogStorageDir` directive)
14. Audit log offset
15. Audit log size
16. Audit log hash (the has begins with the name of the algorithm used, followed by a colon, followed by the hexadecimal representation of the hash itself); this hash can be used to verify that the transaction was correctly recorded and that it hasn't been modified since.

> **Note**
>
> Lines in the index file will be up to 3980 bytes long, and the information logged will be reduced to fit where necessary. Reduction will occur within the individual fields, but the overall format will remain the same. The character `L` will appear as the last character on a reduced line. A space will be the last character on a line that was not reduced to stay within the limit.

Transport Protocol

Audit logs generated in multi-sensor deployments are of little use if left on the sensors. More commonly, they will be transported to a central logging server using the transport protocol described in this section:

1. The transport protocol is based on the HTTP protocol.
2. The server end is an SSL-enabled web server with HTTP Basic Authentication configured.
3. Clients will open a connection to the centralisation web server and authenticate (given the end-point URI, the username and the password).
4. Clients will submit every audit log in a single `PUT` transaction, placing the file in the body of the request and additional information in the request headers (see below for details).
5. Server will process each submission and respond with an appropriate status code:
   a. 200 (OK) - the submission was processed; the client can delete the corresponding audit log entry if it so desires. The same audit log entry must not be submitted again.
   b. 409 (Conflict) - if the submission is in invalid format and cannot be processed. The client should attempt to fix the problem with the submission and attempt

delivery again at a later time. This error is generally going to occur due to a programming error in the protocol implementation, and not because of the content of the audit log entry that is being transported.

c. 500 (Internal Server Error) - if the server was unable to correctly process the submission, due to its own fault. The client should re-attempt delivery at a later time. A client that starts receiving 500 reponses to all its submission should suspend its operations for a period of time before continuing.

> **Note**
>
> Server implementations are advised to accept all submissions that correctly implement the protocol. Clients are unlikely to be able to overcome problems within audit log entries, so such problems are best resolved on the server side.

> **Note**
>
> When en error occurs, the server may place an explanation of the problem in the text part of the response line.

Request Headers Information

Each audit log entry submission must contain additional information in the request headers:

1. Header `X-Content-Hash` must contain the audit log entry hash. Clients should expect the audit log entries to be validated against the hash by the server.
2. Header `X-ForensicLog-Summary` must contain the entire concurrent format index line.
3. The `Content-Lenght` header must be present and contain the length of the audit log entry.

Index