ModSecurity Rule Writing Workshop

Companion to ModSecurity Handbook (WORK IN PROGRESS)

Ivan Ristić

www.feistyduck.com

Introduction

When I started to plan *ModSecurity Handbook*, I had two goals in mind: document every aspect of ModSecurity and teach you how to use it at the same time. As with every other book, I had to choose what will go in and what won't go in. Early on in the writing process it became apparent that there is more to write about ModSecurity than originally anticipated. (In the end, the book was about 120 pages longer than planned.) It also became apparent that the application security space is huge and that there it is not possible to cover ModSecurity and application security in the same book and in a meaningful way.

This document is designed to bridge that gap by showing a number of rules designed to deal with real-life requirements. Whereas *ModSecurity Handbook* will teach you how to write rules on a macro level, this workshop focuses on individual rules in the application security context. Here, unlike in a book, I will be able to mention application security problems without explaining the issues fully (or at all), and trusting that you will seek further documentation should you need it. It's something I could never do in the book.

> **Note**
>
> All the rules presented here are useful, but they are not written to make a rule set. I will often present several rules that perform the same function but in a slightly different way. Furthermore, in some environments these rules could cause either too many false positives or too many false negatives.

HTTP

This section contains the rules that deal with HTTP in some form or another.

Request Method Validation

The HTTP specification documents the main request methods, but allows applications to add new ones. Web servers will often be very lax about what they accept, but there is no reason why your applications should respond to invalid request methods. The following rule will verify that a request method satisfies the basic rules:

```
# Validate request method
SecRule REQUEST_METHOD "!@rx ^[A-Z]{3,16}$" \
    "phase:1,t:none,block,msg:'Invalid request method',logdata:%{REQUEST_METHOD}"
```

Detecting Proxy Requests

Most web server deployments do not work as forward proxies, but may have the forward proxy functionality enabled by default. Use the following rule to detect `CONNECT` requests, which are typically used for SSL tunneling:

```
# Detect CONNECT method
SecRule REQUEST_METHOD "@streq CONNECT" \
    "phase:1,t:none,block,msg:'Proxy access attempt (CONNECT)'"
```

Anyone with a web server on the Internet will receive probes from the user agents looking for open proxies. Such requests differ from all other requests because they contain absolute URIs. The HTTP protocol specifies that absolute URIs (those that contain hostname information) are only to be used with

forward proxies, but you may find that some automated tools still use them. Web servers will normally accept them, even if they do not work as proxies.

The following rule will detect an absolute request URI:

```
# Detect absolute URI
SecRule REQUEST_URI_RAW "@rx (?i)^(ht|f)tp://" \
    "phase:1,t:none,pass,msg:'Proxy access attempt (absolute URI)'"
```

The previous rule may produce false positives if you have legitimate user agents who use this request URI style. You can minimize the false positives by enumerating the hostnames to which your servers are expected to respond, and detecting those requests that specify something else.

Restricting Request Methods

With most applications you will be able to restrict the allowed request methods and significantly reduce the attack surface.

Most applications will use the basic four methods, as allowed by the following rule:

```
# Accept only commonly used request methods
SecRule REQUEST_METHOD "!@rx ^(?:GET|HEAD|POST|OPTIONS)$" \
    "phase:1,t:none,block,msg:'Only standard request methods allowed',logdata:%{REQUEST_METHOD}"
```

Some applications may be using REST, in which case you may need to allow the `PUT` and `DELETE` methods:

```
# Accept only commonly used request methods + REST
SecRule REQUEST_METHOD "!@rx ^(?:GET|HEAD|POST|OPTIONS|PUT|DELETE)$" \
    "phase:1,t:none,block,msg:'Only standard request methods allowed',logdata:%{REQUEST_METHOD}"
```

The following rule can be used with applications that require full WebDAV support:

```
# Accept only commonly used request methods + WebDAV
SecRule REQUEST_METHOD "!@rx ^(?:GET|HEAD|POST|OPTIONS|PROPFIND|PROPPATCH|MKCOL|PUT|DELETE|\
COPY|MOVE|LOCK|UNLOCK)$" \
    "phase:1,t:none,block,msg:'Only standard and WebDAV request methods allowed',\
    logdata:%{REQUEST_METHOD}"
```

Request Protocol Validation

There are only two protocol versions that your web servers should ever see—HTTP 1.0 and HTTP 1.1. Invalid protocol versions may be used as evasion attacks and for web server fingerprinting. The following rule allows only the two aforementioned protocol versions:

```
# Allow only HTTP/1.0 and HTTP/1.1
SecRule REQUEST_PROTOCOL "!@rx (?i)^HTTP/1\.[01]$" \
    "phase:1,t:none,block,msg:'Invalid request protocol',logdata:%{REQUEST_PROTOCOL}"
```

Restricting Content Types

Restricting allowed content types is an important aspect of the attack surface reduction approach. If you can find out what content types are allowed, you can ensure that you allow only those through:

```
SecRule REQUEST_HEADERS:Content-Type "!@rx \
    (?i)^(multipart/form-data|application/x-www-form-urlencoded)" \
```

```
    "phase:1,t:none,block,msg:'Content type not allowed',\
    logdata:'%{MATCHED_VAR}'"
```

If you are not sure, deploy the rule initially to only warn, and observe the alerts.

Disable File Uploads

Sometimes you won't know what the allowed content types are for an application, and you may not be able to deploy a positive-security based rule. However, you may be able to find out if the application uses file uploads. If it does not, it would make good sense to disable them altogether:

```
# Do not allow multipart/form-data content type, which is used for file uploads
SecRule REQUEST_HEADERS:Content-Type "@rx (?i)^multipart/form-data" \
 "phase:1,t:none,block,msg:'Content type multipart/form-data not allowed'"
```

Restrict Request Headers

If you want to be aware of what request headers are used in your application, you could write a rule to alert you on unknown headers. For example:

```
SecRule REQUEST_HEADERS_NAMES "!@rx (?i)^(\
Accept|\
Accept-Charset|\
Accept-Encoding|\
Accept-Language|\
Accept-Ranges|\
Authorization|\
Cache-Control|\
Cookie|\
Cookie2|\
Connection|\
Content-Encoding|\
Content-Language|\
Content-Length|\
Content-Location|\
Content-MD5|\
Content-Type|\
Date|\
Expect|\
From|\
Host|\
If-Match|\
If-Modified-Since|\
If-None-Match|\
If-Range|\
If-Unmodified-Since|\
Keep-Alive|\
Pragma|\
Range|\
Referer|\
TE|\
Trailer|\
Transfer-Encoding|\
UA-CPU|\
User-Agent|\
Via\
```

```
)$" \
"phase:1,t:none,pass,msg:'Unknown request header'"
```

But be aware that, on a public Internet, this sort of rule will produce many false positives over time. You may be able to use it in an internal environment.

> **Note**
>
> The previous rule uses an unoptimized regular expression to allow for easy modification. If you want to optimize it for production, use the optimization as explained in the section "Optimized Regular Expression Pattern" in Chapter 10.

Validate URL Encoding

Invalid URL encoding is often used as an evasion technique against the applications and platforms that accept (and possibly process) invalid URL encoded pairs. Some applications that construct URL in JavaScript may omit to encode data and cause false positives, but that happens only rarely.

Because ModSecurity will automatically decode URL-encoded content where appropriate, you are limited to how you are able to detect invalid URL encodings. The use of the `@validateUrlEncoding` operator is only appropriate with raw input data. One such variable is `REQUEST_URI_RAW`, which enables you to discover invalid URL encodings anywhere in the request URI (that includes any parameters submitted in the query string):

```
# Detect invalid URL encoding in request URI
SecRule REQUEST_URI_RAW @validateUrlEncoding \
    "phase:1,pass,t:none,msg:'Invalid URL encoding in request URI'"
```

The previous rule is not able to detect invalid URL encodings for the data transported in request bodies. Because the request body will be available in `REQUEST_BODY`, you may be able to look at it for invalid encodings. ModSecurity will decode valid URL encodings but leave the invalid ones intact. However, it is not possible to distinguish between an invalid URL encoding and a legitimate `%` placed in input.

The version of ModSecurity in the trunk will raise a flag if it encounters invalid URL encodings anywhere, covering both the query string and the request body parameters. The rule is simply:

```
# Detect invalid URL encoding
SecRule URLENCODED_ERROR "@eq 1" \
    "phase:2,pass,t:none,msg:'Invalid URL encoding detected'"
```

IIS %u Encoding

[...]

Host is an IP Address

Seeing an IP address in the `Host` request header is a pretty good indication of a worm or automated exploit, although there's a small number of sites that do not use domain names. If you can verify that your sites do not (use IP addresses for access), the following rule will detect a very large class of attacks and reconnaissance requests:

```
SecRule REQUEST_HEADERS:Host "^[\d\.]+$" \
    "phase:1,t:none,block,msg:'Host is an IP address'"
```

Evasion Techniques

[...]

NUL Bytes

NUL bytes, raw or encoded, are often used to truncate attack payloads or interfere with the concatenation operations in application code. They have no known legitimate use in web applications. The following rule will detect NUL bytes in request parameters:

```
SecRule ARGS "@validateByteRange 1-255" \
    "phase:2,t:none,msg:'NUL byte(s) detected'"
```

Full-Width/Half-Width Evasion

[...]

```
SecRule ARGS "@rx %u[fF]{2}[0-9a-fA-F]{2}" \
    "phase:2,t:none,block,msg:'Full-Width/Half-Width Unicode Evasion Attempt'"
```

UTF-7

[...]

```
SecRule ARGS "@rx \+A\w+-" \
    "phase:2,t:none,block,msg:'UTF-7 encoding detected'"
```

User Agents

[...]

Bad User Agents

Some user agents are known to be bad and they don't even try to hide themselves. That makes detection very easy:

```
SecRule REQUEST_HEADERS:User-Agent "@pmFromFile bad-user-agents.txt" \
    "phase:1,t:none,block,msg:'Bad user agent'"
```

The file `bad-user-agents.txt` should contain the list of unwanted user agents, for example:

```
morfeus fucking scanner
toata dragostea mea pentru diavola
made by zmeu
plesk
revolt
zmeu
morfeus strikes again
```

Attacks

[...]

Remote File Inclusion

Remote file inclusion (RFI) is an attack that can be executed only against applications written in PHP, which is able to retrieve remote code and execute it. (The `allow_url_fopen` and `allow_url_include` configuration options can be used to disable this functionality.)

RFI attacks are generally easy to detect, although some sites may produce a high rate of false positives:

```
SecRule ARGS "@rx (?i)^(f|ht)tps?://([^/])" \
    phase:2,t:none,msg:'Remote file inclusion (RFI) attack'
```

Local File Inclusion

Local file inclusion (LFI) attacks can be used for two purposes:

- Information leakage, by forcing applications to read a local file and include its content in HTML pages.
- Code execution, by forcing applications to execute code from a file on the local filesystem.

When remote file inclusion attacks fail, attackers often turn to local file inclusion (LFI). LFI attacks are more difficult to use for code execution because the attacker must first find a way to place the code on the local filesystem (and in a known location). The placement is usually done via file upload, or by injecting code into log files.

Local file inclusion is less obvious and thus more difficult to detect. The following detection strategies can be used:

- Detect directory back-references (e.g., as in `/../../../../../../../../../../etc/passwd`)
- Detect directory self-references (e.g., as in `/etc/./passwd`)
- Detect common sensitive file names (e.g., `/etc/passwd`, `/var/log/httpd/error_log`, `/var/log/apache2/error_log`, etc)

```
# TODO
```

PHP Streams Code Injection

In addition to RFI and LFI attacks, there are other ways to attack incorrectly configured PHP applications. The attacks using PHP stream wrappers are as dangerous, but not as well known. They are reasonably easy to detect, as below:

```
SecRule ARGS "@rx (?i)^php://" \
    "phase:2,t:none,msg:'PHP php:// code injection'"

SecRule ARGS "@rx (?i)^data:" \
    "phase:2,t:none,msg:'PHP data: code injection'"
```

SQL Injection

[...]

Cross-Site Scripting

[...]

Code Injection

This section focuses on code injection attacks, where a vulnerable system is tricked into running the code supplied by the attacker.

PHP Code Injection

PHP code cannot be injected directly, but it may be possible to have the code recorded on disk only to be executed later using a local inclusion attack.

The following rule will detect a code injection attempt, but ignore XML documents, which use similar syntax:

```
SecRule ARGS "@rx <\?(?!xml)"
```

> **Note**
>
> It is possible to configure PHP to use the <% and %> syntax, but this feature is disabled by default.

SSI Injection

Server side includes (SSI) is a technology that allows dynamic content to be added to otherwise static pages. SSI injection attacks are targeted at applications that might use some of the information provided to generate pages that will be served through a web server. If the web server has SSI enabled, it might be possible for an attacker to inject SSI code and have it executed.

The following rule will detect an attempt to inject SSI code:

```
SecRule ARGS|REQUEST_HEADERS "@rx (?i)<!--\W*?#\W*?(?:echo|exec|printenv|include)" \
    "phase:2,t:none,pass,msg:'SSI injection'"
```

In addition to looking at request parameters, we look at request headers, because SSI injection is frequently targeted at web statistics software that will generate HTML pages that contain `User-Agent` and `Referer` information.

Information Leakage

Information leakage occurs when a system is configured to reveal too much information about itself, by design or by mistake. Such leakage may or may not be obviously sensitive, but even the smallest piece of information can sometimes help the attacker advance in his attacks.

Apache Server Header

Apache treats the Server response header as a special case and injects it into responses just before the headers are sent. (It gives the Date header the same treatment, by the way.) As a result of that special treatment, ModSecurity is not able to inspect the contents of the Server header in embedded mode.

In reverse proxy mode ModSecurity is able to inspect the Server header of the backend web server. When an Apache server is misconfigured to leak information the Server response headers could look like in the following example:

```
Server: Apache/2.2.9 (Debian) PHP/5.2.6-1+lenny4 with Suhosin-Patch
Server: Apache/2.2.9 (Debian)
```

The ideal configuration (which you get when you set ServerTokens to ProductOnly) is this:

```
Server: Apache
```

The rule to detect information leakage will warn if the Server response header begins with Apache and has at least one character following:

```
# Apache Server header information leakage
SecRule RESPONSE_HEADERS:Server "@rx ^Apache.+" \
    "phase:3,t:none,pass,msg:'Apache Server header information leakage'"
```

In reverse proxy mode, it is not enough to configure the proxy itself to send minimal server information. You also have to use mod_headers to override whatever Server response header is sent by backend servers. For example:

```
Header set Server Apache
```

X-Powered-By Information Leakage

Some application servers add version information in an X-Powered-By header of every response they generate.

```
SecRule &RESPONSE_HEADERS:X-Powered-By "!@eq 0" \
    "phase:3,t:none,pass,msg:'Response header information leakage (X-Powered-By)'"
```

Set the configuration parameter expose_php to Off to stop PHP from leaking version information. Alternatively, use mod_headers to remove the X-Powered-By header:

```
Header unset X-Powered-By
```

ASP.NET Version Leakage

ASP.NET will add version information in an X-AspNet-Version header of every response it generates.

```
SecRule &RESPONSE_HEADERS:X-AspNet-Version "!@eq 0" \
    "phase:3,t:none,pass,msg:'Response header information leakage (X-AspNet-Version)'"
```

You can use mod_headers to remove the X-AspNet-Version header:

```
Header unset X-AspNet-Version
```

PHP phpinfo() Output Leakage

The `phpinfo()` output in PHP is very useful to inspect PHP configuration, but it is often left by mistake in production web sites. The output may also be a sign of a successful code injection attack. The following rule will detect the use of `phpinfo()` anywhere on a site (assuming response buffering is enabled, of course):

```
# PHP phpinfo() output detection
SecRule RESPONSE_BODY "@rx <title>phpinfo\(\)</title>" \
    "phase:4,log,pass,t:none,msg:'PHP phpinfo() output detected'"
```

PHP and Suhosin Presence Leakage

When used with the configuration parameter `expose_php` set to `On`, PHP will use images in the output of the `phpinfo()` function. These images can be retrieved even when `phpinfo()` is not used, and there is at least one easter egg image. Suhosin adds a further one image.

It may be interesting to have rules to detect access to these images:

```
SecRule QUERY_STRING "@rx (?i)^=php" \
    "phase:1,t:none,pass,msg:'PHP embedded image access'"

SecRule QUERY_STRING "@rx (?i)^=suho" \
    "phase:1,t:none,pass,msg:'PHP/Suhosin embedded image access'"
```

The previous rules will trigger if `phpinfo()` with images is used anywhere on the site, but it may also signal an advanced reconnaissance effort.