AFP Rule Writing Guide

Technical Note

VERSION: 1.0

UPDATED: NOVEMBER 2014

Copyright Notices

Table of Contents

1 Introduction

Application Firewall Pack (AFP) services are natively integrated in the KEMP LoadMaster. This enables secure deployment of web applications, preventing Layer 7 attacks while maintaining core load balancing services which ensures superior application delivery and security. AFP functionality directly augments the LoadMaster's existing security features to create a layered defence for web applications - enabling a safe, compliant and productive use of published services.

If you have an AFP license and AFP Support, KEMP provides a number of commercial rules, such as **ip_reputation**, which can be set to automatically download and update on a daily basis. These commercial rules are targeted to protect against specific threats. The KEMP-provided commercial rules are available when signed up to an AFP subscription.

You can also upload other rules such as the ModSecurity core rule set which contains generic attack detection rules that provide a base level of protection for any web application.

You can also write and upload your own custom rules, if required.

With the AFP-enabled LoadMaster, you can choose whether to use KEMP-provided rules, custom rules which can be uploaded or a combination of both.

For a more detailed overview of the AFP feature, please refer to the AFP section in the **KEMP LoadMaster, Product Overview**.

For instructions on how to configure the various AFP options in the LoadMaster, refer to the **AFP, Feature Description**.

1.1 Document Purpose

The purpose of this document is to provide some guidance on how to write your own custom AFP rules. These custom rules can be uploaded to the LoadMaster and assigned to Virtual Services as needed.

1.2 Intended Audience

This document is intended to be read by anyone who is interested in finding out more about how to write custom AFP rules.

2 ModSecurity Rule Writing

The ModSecurity Reference Manual should be consulted in any cases where questions arise relating to the syntax of commands:
https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual

In terms of rule writing, the main directive to know is SecRule, which is used to create rules and thus does most of the work.

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

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

The three parts are explained in the sections below.

2.1 VARIABLES

This specifies which places to check in a HTTP transaction. Examples of variables include:

- **ARGS** – all arguments including the POST payload
- **REQUEST_METHOD** – request method used in the transaction
- **REQUEST_HEADERS** – can be used as either a collection of all of the request headers or can be used to inspect selected headers
- Etc. The full list of variables is available here: https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual#Variables

2.2 OPERATOR

This specifies a regular expression, pattern or keyword to be checked in the variable(s). Operators begin with the @ character. The full list of operators is available here:
https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual#Operators

2.3 ACTIONS

This specifies what to do if the rule matches. Actions are defined in seven categories, listed below:

- **Disruptive** – used to allow ModSecurity to take an action, for example allow or block
- **Flow** – affect the flow, for example skip
- **Meta-data** – used to provide more information about rules
- **Variable** – used to set, change and remove variables
- **Logging** – used to influence the way logging takes place
- **Special** – used to provide access to another class of functionality
- **Miscellaneous** – contain actions that do not belong in any other groups.

If no actions are provided, default actions apply as per `SecDefaultAction` (`phase:2,log,auditlog,pass`). The full list of actions are available here:

https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual#Actions

2.4 Rule Syntax

The following rule looks at the request Uniform Resource Identifier (URI) and tries to match the regular expression pattern <script> against it. The double quotes are used because the second parameter contains a space:

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

To split a long line into two, use a single backslash character, followed by a new line:

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

Multiple variables can be used in a rule as long as they are separated using the pipe character, for example:

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

The **SecDefaultAction** directive is used if no actions are defined for a rule. For example, the following rule:

```
SecRule ARGS D1
```

Is equivalent to:

```
SecRule ARGS D1 phase2:log:auditlog,pass
```

2.4.1 Rule Example 1 – Cross Site Scripting (XSS) Attack

The following rule is used to avoid XSS attacks by checking for a <script> pattern in the request parameters and header and generates an 'XSS Attack' message with a 404 status response.

```
SecRule ARGS|REQUEST_HEADERS "@rx <script>" id:101,msg: 'XSS
Attack',severity:ERROR,deny,status:404
```

2.4.1.1 Variables

Details about the variables in this rule example are in the table below:

Variable	Definition
ARGS	Request parameters
REQUEST_HEADERS	All of the request headers

2.4.1.2 Operator

"@rx <script>" – Performs a regular expression match of the pattern (in this case <script>) provided as a parameter.

2.4.1.3 Actions

Details of the actions contained in this rule example are provided in the table below:

Action(s)	Description
`id, msg, severity, deny, status`	These are all of the actions to be performed if the pattern is matched.
`id:101`	The unique ID that is assigned to the rule (or chain) in which it appears.
`msg: "XSS Attack"`	The custom message (i.e. XSS Attack) assigned to the rule (or chain) in which it appears.
`Severity:ERROR`	The severity of the rule. Severities include: – EMERGENCY (0) – ALERT (1) – CRITICAL (2) – ERROR (3) – WARNING (4) – NOTICE (5) – INFO (6) – DEBUG (7)
`deny`	This stops rule processing and intercepts transaction. This is a disruptive action.
`status:404`	This specifies the response status code (404) with actions deny and redirect.

2.4.2 Rule Example 2 – Whitelist IP Address

The following example shows how to whitelist an IP address to bypass the ModSecurity engine:

```
SecRule REMOTE_ADDR "@ipMatch 192.168.1.101" \
      id:102,phase:1,t:none,nolog,pass,ctl:ruleEngine=off
```

2.4.2.1 Variables

Variable Name: `REMOTE_ADDR`

Variable Definition: The IP address of the remote client

2.4.2.2 Operator

`"@ipMatch 192.168.1.101"` – Performs an IPv4 or IPv6 match of the REMOTE_ADDR variable data. In this care – this is the whitelisted IP address.

2.4.2.3 Actions

Action(s)	Description
`id:101`	The unique ID that is assigned to the rule (or chain) in which it appears.
`phase:1`	Places the rule (or chain) in Phase 1 processing. There are five phases, including: – Request Headers (1) – Request Body (2) – Response Headers (3) – Response Body (4) – Logging (5)
`t:none`	Indicates that no action is used to transform the value of the variable used in the rule before matching. For example, `t:utf8toUnicode` converts all UTF-8 character sequences to Unicode to assist in input normalization.
`nolog`	Prevents rule matches from appearing in both the error and audit logs.
`pass`	Continues processing with the next rule in spite of a successful match.
`ctl:ruleEngine=off`	This action changes ModSecurity configuration on a transient, per-transaction basis. This only affects the transaction in which the action is executed. In this case, the ModSecurity rule engine is turned off.

2.4.3 Rule Example 3 – Chaining Rules

This section shows an example of chaining two rules. In this example, the first rule checks if the username (`ARGS:username`) for the string admin (`streq admin`) using a string comparison. If the first rule holds true, the second rule is activated which denies all requests that are not from the REMOTE_ADDR 192.168.1.111 IP Address (`!streq 192.168.1.111`).

```
SecRule ARGS:username "@streq admin" chain,deny
SecRule REMOTE_ADDR "!streq 192.168.1.111"
```

2.4.4 Rule Example 4 – Shellshock Bash Attack

This section shows an example of the rules requires to mitigate the Shellshock Bash attack. There are two rules needed in this case. Details of both rules are provided in the sections below.

2.4.4.1 First Rule

This is the first rule:

```
SecRule REQUEST_LINE|REQUEST_HEADERS|REQUEST_HEADERS_NAMES
"@contains () {"
"phase:1,id:'2100080',block,t:none,t:utf8toUnicode,t:urlDecodeUni
,t:compressWhitespace,msg:'SLR: Bash ENV Variable Injection
Attack',tag:'CVE-2014-6271',tag:'http://cve.mitre.org/cgi-
bin/cvename.cgi?name=CVE-2014-
6271',tag:'https://securityblog.redhat.com/2014/09/24/bash-
specially-crafted-environment-variables-code-injection-attack/'"
```

2.4.4.1.1 Variables

Details about the variables in this example rule are provided in the table below:

Variable	Definition
REQUEST_LINE	This variable holds the complete request line sent to the server (including the request method and HTTP version information).
REQUEST_HEADERS	All of the request headers
REQUEST_HEADERS_NAMES	All of the names of the request headers.

2.4.4.1.2 Operator

"@contains () {" – Checks the REQUEST_LINE|REQUEST_HEADERS|REQUEST_HEADERS_NAMES variables for the string '() {' and returns true if found.

2.4.4.1.3 Actions

Action(s)	Description
phase:1	Places the rule (or chain) in Phase 1 processing. There are five phases, including: – Request Headers (1) – Request Body (2) – Response Headers (3) – Response Body (4) – Logging (5)
id:'2100080'	The unique ID that is assigned to this rule (or chain) in which it appears.

Action(s)	Description
`block`	This performs the disruptive action defined by the previous SecDefaultAction. This allows rule writers to request a blocking action without specifying how the blocking is to be done. The `SecRuleUpdateActionById` directive allows you to override how a rule handles blocking. Please refer to **Appendix B – Rule Block Function** for further details.
`t:none`	Indicates that no action is used to transform the value of the variable used in the rule before matching.
`t:utf8toUnicode`	Converts all UTF-8 character sequences to Unicode to assist in input normalization.
`t:urlDecodeUni`	Decodes a URL-encoded input string with support for the Microsoft-specific %u encoding.
`t:compressWhitespace`	Converts any of the whitespace characters (0x20, \f, \t, \n, \r, \v, 0xa0) to spaces (ASCII 0x20), compressing multiple consecutive space characters into one.
`msg:'SLR: Bash ENV Variable Injection Attack',tag:'CVE-2014-6271'`	The custom message (i.e. XSS Attack) assigned to the rule (or chain) in which it appears.
`tag:'http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271'` `tag:'https://securityblog.redhat.com/2014/09/24/bash-specially-crafted-environment-variables-code-injection-attack/'`	Assigns a tag (category) to a rule (or chain). This is metadata allows easy automated categorization of events. Multiple tags can be specified on the same rule.

2.4.4.2 Second Rule

The second rule is as follows:

```
SecRule REQUEST_BODY "@contains () {"
"phase:2,id:'2100081',block,t:none,t:utf8toUnicode,t:urlDecodeUni
,t:compressWhitespace,msg:'SLR: Bash ENV Variable Injection
Attack',tag:'CVE-2014-6271',tag:'http://cve.mitre.org/cgi-
bin/cvename.cgi?name=CVE-2014-
6271',tag:'https://securityblog.redhat.com/2014/09/24/bash-
specially-crafted-environment-variables-code-injection-attack/'"
```

2.4.4.2.1 Variables

Variable Name: `REQUEST_BODY`

Variable Definition: All of the request body.

2.4.4.2.2 Operator

"`@contains () {`" – Checks the `REQUEST_BODY` variable for the string '`() {`' and returns true if found.

2.4.4.2.3 Actions

Action(s)	Description
`phase:2`	Places the rule (or chain) in Phase 2 processing. There are five phases, including: – Request Headers (1) – Request Body (2) – Response Headers (3) – Response Body (4) – Logging (5)
`id:'2100081'`	The unique ID that is assigned to this rule (or chain) in which it appears.
`block`	This performs the disruptive action defined by the previous SecDefaultAction. This allows rule writers to request a blocking action, but without specifying how the blocking is to be done. The `SecRuleUpdateActionById` directive allows you to override how a rule handles blocking. Please refer to **Appendix B – Rule Block Function** for further details.
`t:none`	Indicates that no action is used to transform the value of the variable used in the rule before matching.
`t:utf8toUnicode`	Converts all UTF-8 character sequences to Unicode to assist in input normalization.
`t:urlDecodeUni`	Decodes a URL-encoded input string with support for the Microsoft-specific %u encoding.
`t:compressWhitespace`	Converts any of the whitespace characters (0x20, \f, \t, \n, \r, \v, 0xa0) to spaces (ASCII 0x20), compressing multiple consecutive space characters into one.
`msg:'SLR: Bash ENV Variable Injection Attack',tag:'CVE-2014-6271'`	The custom message (i.e. XSS Attack) assigned to the rule (or chain) in which it appears.
`tag:'http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271'` `tag:'https://securityblog.redhat.com/2014/09/24/bash-specially-crafted-environment-variables-code-injection-attack/'`	Assigns a tag (category) to a rule (or chain). This is metadata which allows easy automated categorization of events. Multiple tags can be specified on the same rule.

Appendix A – KEMP WUI Settings

In the LoadMaster Web User Interface (WUI), AFP settings can be configured for each individual Virtual Service.

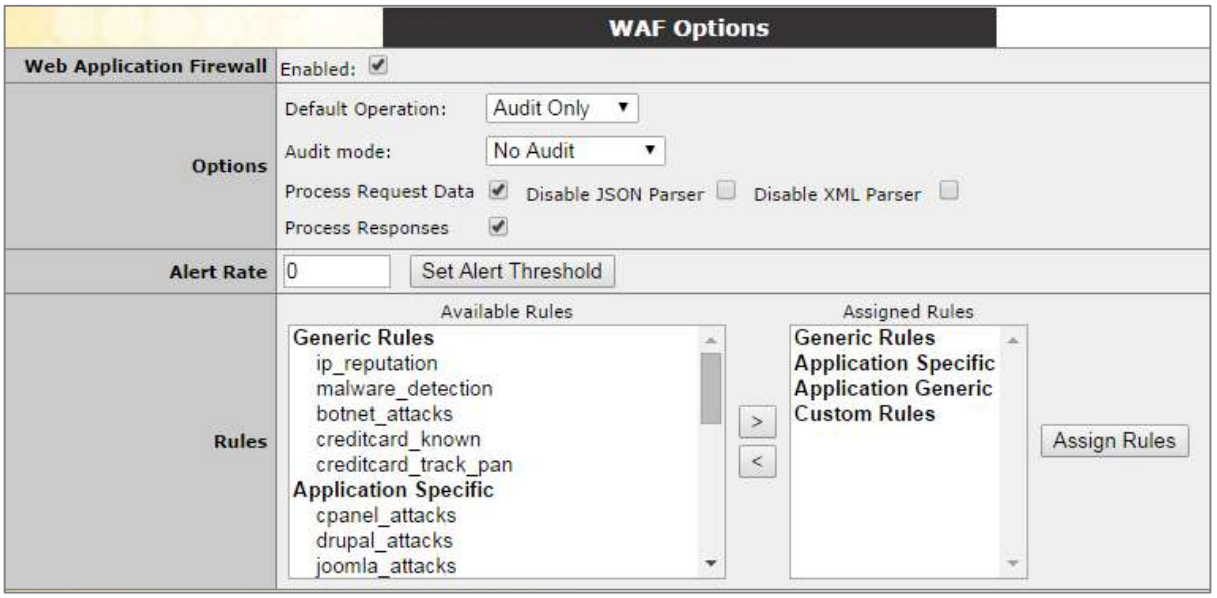

Figure 0-1: WAF Options

In the **WAF Options** section of the Virtual Service modify screen (**Virtual Services > View/Modify Services > Modify**), there is a drop-down list called **Default Operation**. The **Default Operation** can be set to **Audit Only** or **Block Mode**.

The **Audit Only** mode of operation sets the `SecDefaultAction` to `phase:2,log,auditlog,pass`.

The **Block Mode** of operation sets the `SecDefaultAction` to `phase:2,log,auditlog,block,drop`.

Appendix B – Rule Block Function

The rule block function is quite complicated. This section offers further explanation of the rule block function. The following example has been taken from https://github.com/Spiderlabs/ModSecurity/wiki/Reference-Manual#block and further explanatory text has been added.

The `block` action is essentially a placeholder that is intended to be used by rule writes to request a blocking action, but without specifying how the blocking is to be done. The `SecDefaultAction` command specifies how the blocking is to be done. The `block` action is a placeholder that will be replaced by the action from the last `SecDefaultAction` in the same context.

Block Example 1

The following example shows the `SecDefaultAction` set to `deny`. The second rule will "deny" because the `SecDefaultAction` is set to `deny`.

```
SecDefaultAction phase:2,deny,id:101,status:403,log,auditlog
SecRule ARGS attack2 phase:2,pass,id:103
SecRule ARGS attack1 phase:2,block,id:102
```

Block Example 2

The following example shows the usage of the `SecRuleUpdateActionById` command to override how a rule handles blocking. The `SecRuleUpdateActionById` command allows a rule to be reverted back to the previous `SecDefaultAction`. In this example, the first rule (`SecRule ARGS attack1 phase:2,deny,id:1`) would `deny` based on meeting the successful conditions associated with the rule.

By using the `SecRuleUpdateActionById` against rule `Id 1` and indicating `block`, we are associating the first rule action to that of the `SecDefaultAction` which is `pass`. So in the case, the first rule would `pass` based on meeting the successful conditions associated with the rule; it would not `deny`.

```
SecDefaultAction phase:2,pass,log,auditlog
SecRule ARGS attack1 phase:2,deny,id:1
SecRuleUpdateActionById 1 block
```


References

Unless otherwise specified, the following documents can be found at http://www.kemptechnologies.com/documentation.

ModSecurity Reference Manual
https://github.com/SpiderLabs/ModSecurity/wiki/Reference-Manual

AFP, Feature Description

KEMP LoadMaster, Product Overview

Document History

Date	Change	Reason for Change	Version	Resp.
Nov 2014	Initial draft	First draft of document	1.0	LB

