Progress®
Kemp® LoadMaster®

Technical Note WAF Rule Writing Guide

Generated on: 21 April 2026

Copyright

Visit the following page online to see Progress Software Corporation's current Product Documentation Copyright Notice/Trademark Legend: Product Documentation Copyright Notice & Trademarks | Progress

Table of Contents

Chapter 1: Introduction 5

Chapter 2: ModSecurity Rule Writing 6

Variables 7

Operator 7

Transformation Functions 7

Actions. 8

Rule Syntax 8

Rule Example 1 – Cross Site Scripting (XSS) Attack 9

Rule Example 2 – Allow-list an IP Address 11

Rule Example 3 – Chaining Rules 12

Rule Example 4 – Shellshock Bash Attack. 13

WUI Settings 18

Rule Block Function 18

Chapter 3: Managing Custom WAF Rules in the LoadMaster 20

Add a Custom Rule 20

Delete/Download a Custom Rule or Data File. 22

Chapter 4: Assigning Custom Rules to a Virtual Service. 23

WAF Misconfigured State 25

Chapter 5: Backing Up and Restoring WAF Configuration 26

Chapter 6: References 27

1

Introduction

Web Application Firewall (WAF) services are natively integrated in the 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. WAF 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 a WAF license and WAF Support, Progress Kemp provides the industry standard OWASP CRS rule set, which contains generic attack detection rules that provide a base level of protection for any web application. The Progress Kemp-provided and supported rules are available when signed up to a WAF subscription.

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

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

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

For instructions on how to configure the various WAF options in the LoadMaster, refer to the Web Application Firewall, Feature Description.

Document Purpose

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

2

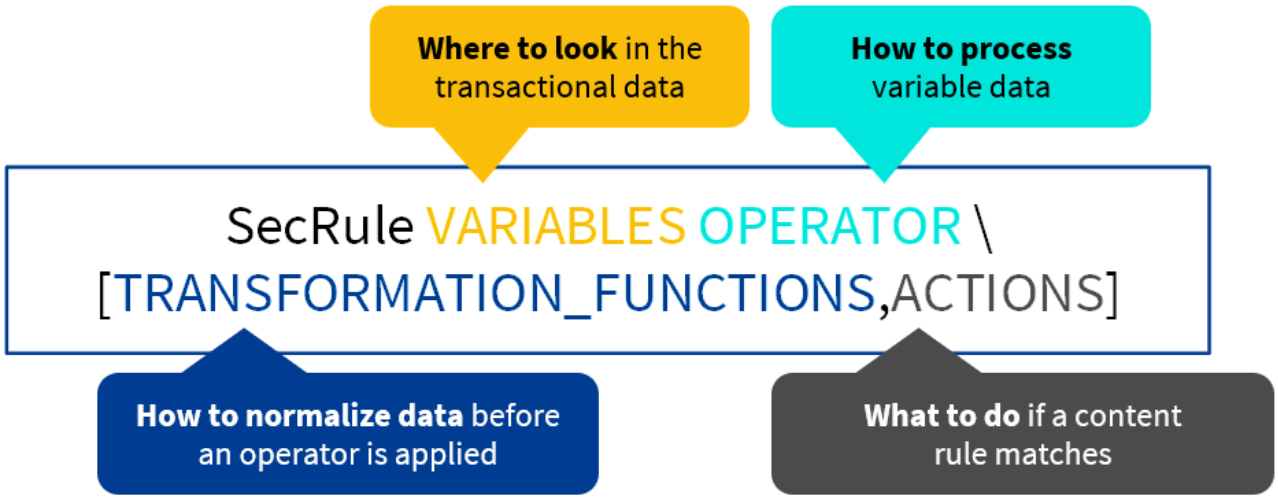
ModSecurity Rule Writing

The ModSecurity Reference Manual should be consulted in any cases where questions arise relating to the syntax of commands.

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 [TRANSFORMATION_FUNCTIONS,ACTIONS]
```



The rule consists of four parts:

- **VARIABLES:** Tells the WAF engine where to look in the transactional data (HTTP request and response data).
- **OPERATOR:** Tells the WAF engine how to process the variable data.
- **TRANSFORMATION_FUNCTIONS:** Tells the WAF engine how to normalize data before an operator is applied.
- **ACTIONS:** Tells the WAF engine what to do if a rule matches.

The four parts are explained in the sections below.

Contents

- Variables
- Operator
- Transformation Functions
- Actions
- Rule Syntax
- WUI Settings
- Rule Block Function

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
- And so on. The full list of variables is available here: ModSecurity Reference Manual

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 under the **Operators** section of Reference Manual on the: ModSecurity Documentation page.

Transformation Functions

There are a number of transformation functions that can be performed, for example:

- Anti-evasion (such as lowercase, normalisePath, removeNulls, replaceComments, compressWhitespace)

- Decoding (such as base64Decode, hexDecode, jsDecode, urlDecodeUni)
- Encoding (such as base64Encode, hexEncode)
- Hashing (such as sha1, md5)

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 is available in the **Actions** section of the Reference Manual on the ModSecurity Documentation page

When constructing rules, you can specify at what phase the rule should run. Specifying the correct phase can be beneficial in order to reduce CPU processing. For more information on phases, refer to the ModSecurity Reference Manual.

Rule Syntax

The rule-writing language is somewhat unusual and can be very difficult to follow and understand. When writing a new custom rule (to keep rules as clear and maintainable as possible), try to follow these best practices:

- Comment clearly **what** the rule is doing and **why**, for future reference
- Put each rule action on a separate line for clarity (use a \ at the end of the line to escape the newline character)
- Indent each rule action for clarity
- Keep the same order of actions across rules to avoid mistakes (1: id, 2: phase, 3: deny/pass, and so on)
- Write a clear rule 'msg' message to add to the log line when the rule matches

An example rule is as follows:

```
#
# -- Validate User-Agent Header
#
# The only legitimate access to our API is by IoT devices that we have deployed
# in the field. These will always identify themselves with a user agent that
# looks like:
```

```
#     AcmeWidget/1.2.34-r56
# Deny any request that *does not* (note the ! to negate the match) send a
# legitimate-looking user agent. Make the match case insensitive by using a
# 't:lowercase' transformation.
#
SecRule REQUEST_HEADERS:User-Agent "!@beginsWith acmewidget/" \
   "id:1000,\
   phase:1,\
   deny,\
   t:none,t:lowercase,\
   msg:'Legitimate AcmeWidget User-Agent header required',\
   logdata:'%{MATCHED_VAR}'"
```

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 "@rx <script>"
```

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

```
SecRule ARGS "@rx D1"
```

Is equivalent to:

```
SecRule ARGS "@rx D1" "phase:2,log,auditlog,pass"
```

Contents

- Rule Example 1 – Cross Site Scripting (XSS) Attack
- Rule Example 2 – Allow-list an IP Address
- Rule Example 3 – Chaining Rules
- Rule Example 4 – Shellshock Bash Attack

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,\
deny,\
status:404,\
msg:'XSS Attack'"
```

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

Operator

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

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

Action(s)	Description
	rule (or chain) in which it appears.
`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.

Rule Example 2 – Allow-list an IP Address

The following example shows how to allow-list an IP address to bypass the ModSecurity engine by switching the engine off for connections from that IP address:

```
SecRule REMOTE_ADDR "@ipMatch 192.168.1.101" \

    "id:102,\

    phase:1,\

    pass,\

    t:none,\

    nolog,\

    ctl:ruleEngine=off"
```

Variables

Variable Name: REMOTE_ADDR

Variable Definition: The IP address of the remote client

Note: Rules listing IP addresses with the /32 subnet mask are not supported and will not upload successfully.

Operator

“@ipMatch 192.168.1.101” – Performs an IPv4 or IPv6 match of the REMOTE_ADDR variable data. In this case, this is the allow-listed IP address.

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.

Rule Example 3 – Chaining Rules

Chained rules allow for more complex rule matches where a number of different VARIABLES are used to create a better rule and to help prevent false positives. In programming language concepts – think of chained rules as somewhat similar to AND conditional statements. The disruptive action 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, for the purposes of the disruptive action, the entire rule chain is negative. The

most unique portion should be specified on the first line – this will reduce the number of “normal” requests that will have to be evaluated against the rest of the chained rule set.

In addition to using a number of different VARIABLES in the one rule, it is also possible to chain more than one rule. Below is 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" \

    "id:103,\

    phase:2,\

    deny,\

    t:none,\

    log,\

    chain"

    SecRule REMOTE_ADDR "!@streq 192.168.1.111"
```

Rule Example 4 – Shellshock Bash Attack

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

First Rule

This is the first rule:

```
SecRule REQUEST_LINE|REQUEST_HEADERS|REQUEST_HEADERS_NAMES "@contains () {" \

    "id:2100080,\

    phase:1,\

    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/'"
```

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.

Operator

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

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)

Action(s)	Description
`id:'2100080'`	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 without specifying how the blocking is to be done. The **SecRuleUpdateActionById** directive allows you to override how a rule handles blocking. Please refer to the Rule Block Function section 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.

Second Rule

The second rule is as follows:

```
SecRule REQUEST_BODY "@contains () {" \

    "id:2100081,\

    phase:2,\

    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/'"
```

Variables

Variable Name: REQUEST_BODY

Variable Definition: All of the request body.

Operator

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

Actions

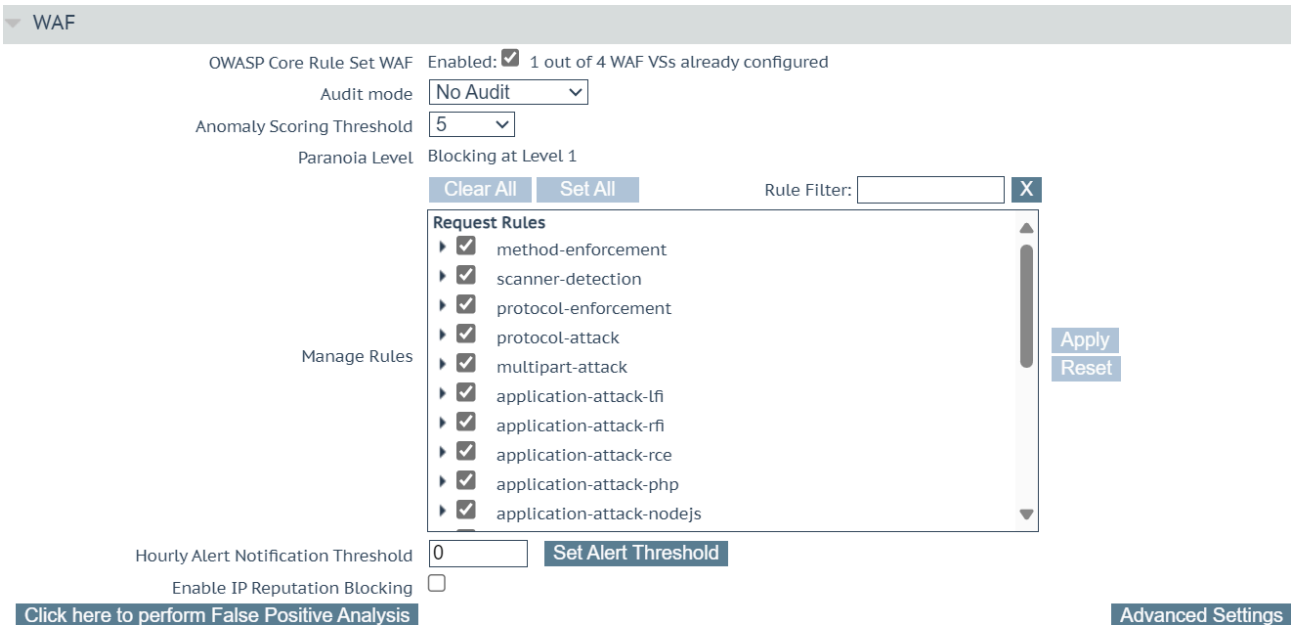
Action(s)	Description
`phase:2`	Places the rule (or chain) in Phase 2 processing. There are five phases, including: • Request Headers (1) • Request Body (2) • Response Headers (3) • Response Body (4)

Action(s)	Description
	• 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 the Rule Block Function section 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.

WUI Settings

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



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 the **Block** topic (under the **Actions** section) of the Reference Manual on the ModSecurity Documentation page and further explanatory text has been added.

The block action is essentially a placeholder that is intended to be used by rule writers 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 this 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
```

3

Managing Custom WAF Rules in the LoadMaster

Refer to the following sections for details on managing custom WAF rules in the LoadMaster.

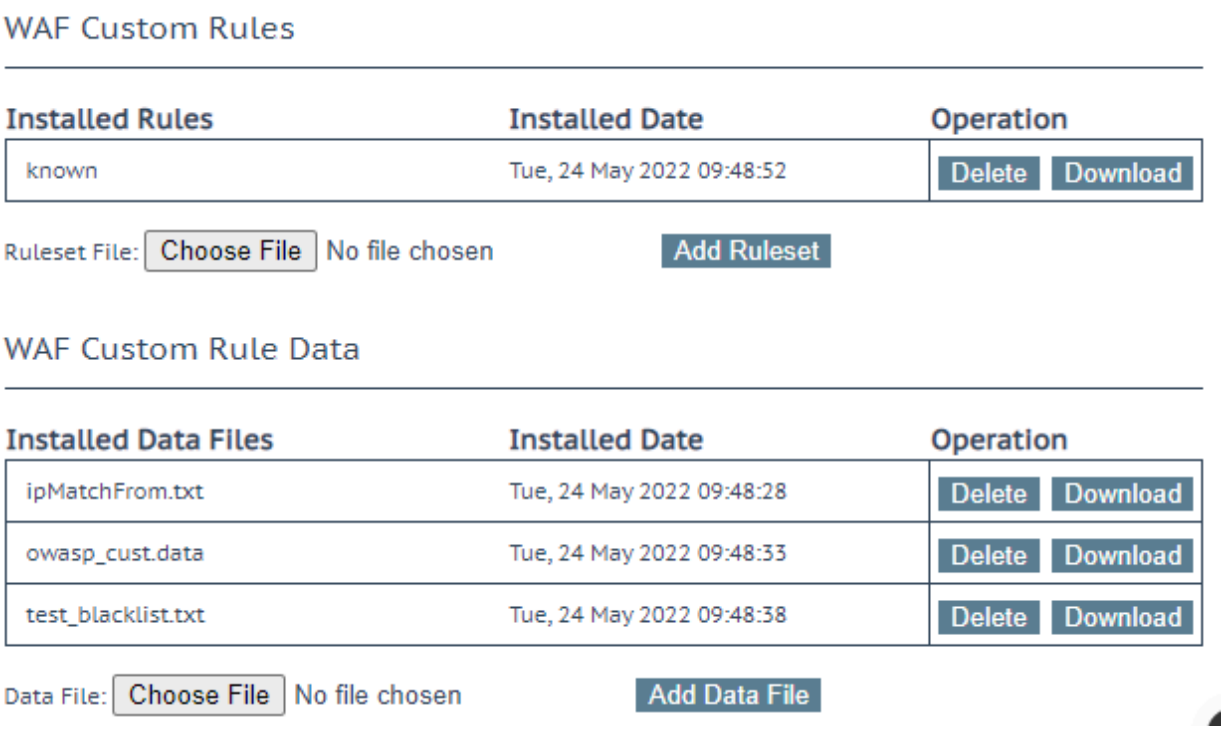
Contents

- Add a Custom Rule
- Delete/Download a Custom Rule or Data File

Add a Custom Rule

Follow the steps below to find out how to add custom WAF rules in the Web User Interface (WUI) of the LoadMaster:

1. In the main menu, select **Web Application Firewall > Custom Rules**.



2. To upload custom rules, click **Choose File** in the **Installed Rules** section.

   Note: Individual rules can be uploaded as .conf files, or you can load a package of rules in a tar.gz file. The first character in the filename must be an alpha character or an underscore (_). The other characters in the filename can include full stops (.) or dashes (-).

   Note: Rules listing IP addresses with the /32 subnet mask are not supported and will not upload successfully.

3. Browse to and select the rules to be uploaded.
4. Click **Add Ruleset**.
5. To upload any additional data files, click **Choose File** in the **Custom Rule Data** section.

   Note: The additional files are for the rules' associated data files. If using a Tarball, the rules and data files can be packaged together.

6. Browse to and select the additional data files.
7. Click **Add Data File**.

The rules will now be available to assign within the Virtual Services modify screen (**Virtual Services > View/ Modify Services > Modify**). Refer to the Assigning Custom Rules to a Virtual Service section to find out how to configure the Virtual Service.

Delete/Download a Custom Rule or Data File



Custom rules and data files can be deleted or downloaded by clicking the relevant buttons.

Note: If a rule is assigned to a Virtual Service, it will not be available for deletion.

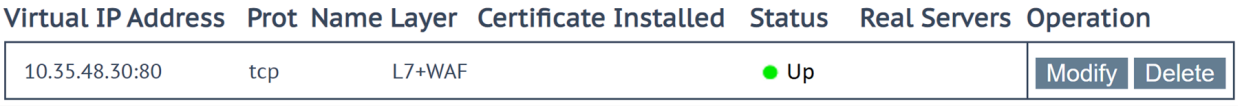
4

Assigning Custom Rules to a Virtual Service

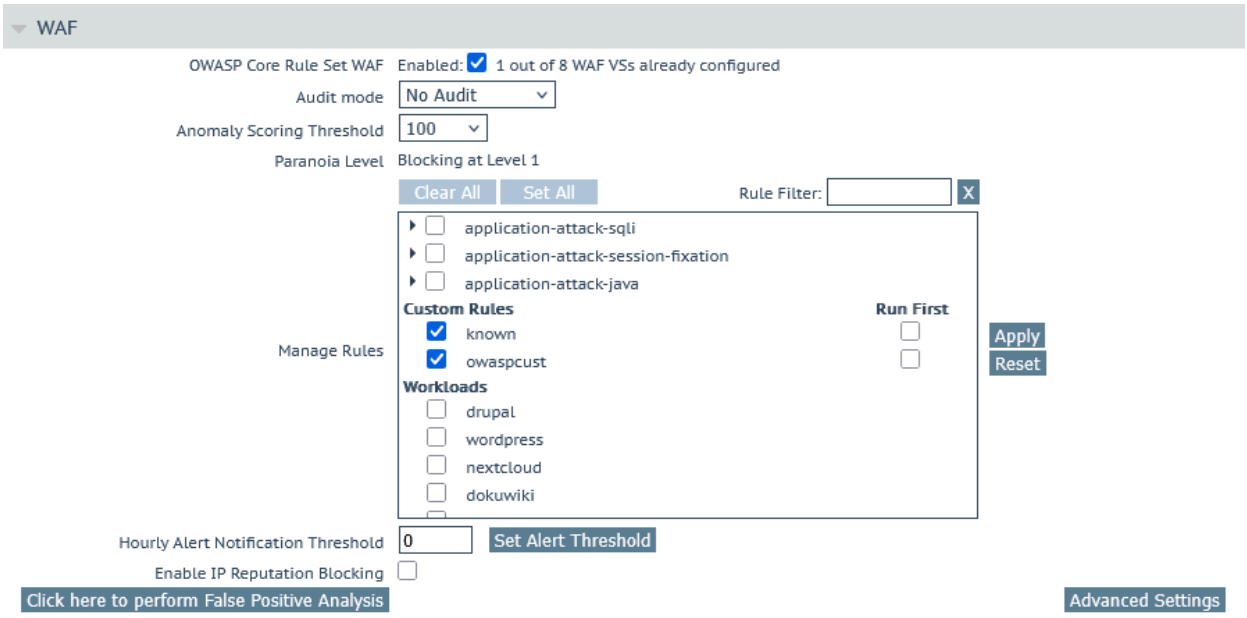
Before you can assign a custom rule to a Virtual Service, the rule must first be installed on the LoadMaster. Refer to the Add a Custom Rule section for instructions on how to do this.

Custom rules can be assigned as needed to each individual Virtual Service. Follow the steps below to assign a custom rule to a Virtual Service:

1. In the main menu of the LoadMaster WUI, select **Virtual Services > View/Modify Services**.



2. Click **Modify** on the relevant Virtual Service.
3. Expand the **WAF** section.



4. Select **Enabled**.

Note: A message is displayed next to the **Enabled** check box displaying how many WAF-enabled Virtual Services exist and it also displays the maximum number of WAF-enabled Virtual Services that can exist. If the maximum number of WAF-enabled Virtual Services has been reached, the **Enabled** check box becomes greyed out.

---

5. Assign Custom Rules by selecting them in the **Manage Rules** section.
6. To enable/disable the individual rules per ruleset, select/unselect the relevant check boxes.Rules can be filtered by entering a filter term in the **Rule Filter** text box.When you have a filter entered, you can do the following:
   - Click **Clear All** to disable all rules for the selected ruleset.
   - Click **Set All** to enable all rules for the selected ruleset.
   - Click **Reset** to disable all rules and rulesets.

Note: There is a **Run First** check box available for custom rules. If the **Run First** check box is enabled for a custom rule, the rule will be run first, before the OWASP Core Rule Set (CRS). If the **Run First** check box is disabled for a custom rule, the custom rule runs after the CRS. The **Run First** check box is disabled by default.

---

7. When finished selecting the relevant rulesets and rules, click **Apply**.

Contents

- WAF Misconfigured State

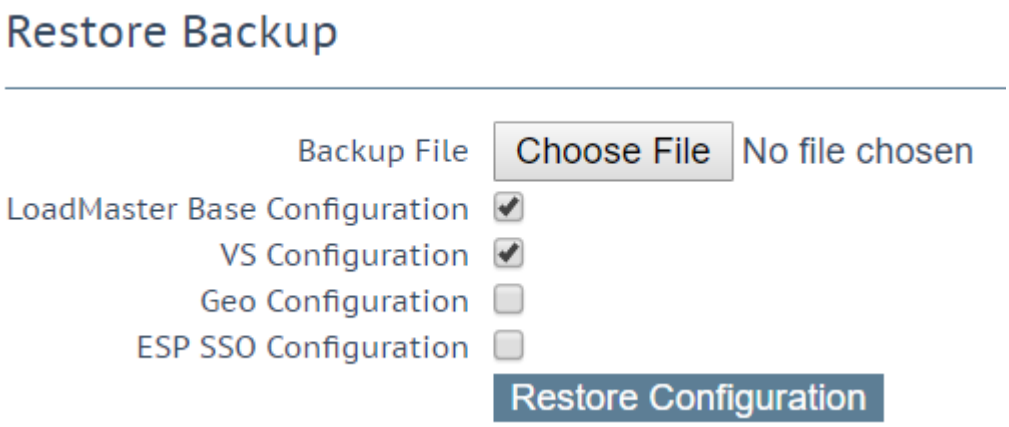
WAF Misconfigured State



On the **View/Modify Services** screen in the LoadMaster WUI, the **Status** of each Virtual Service is displayed. If the WAF for a particular Virtual Service is misconfigured, for example if there is an issue with a rule file, the status changes to **WAF Misconfigured** and turns to red. If the Virtual Service is in this state, all traffic is blocked. WAF can be disabled for that Virtual Service to stop the traffic being blocked, if required, while troubleshooting the problem.

5

Backing Up and Restoring WAF Configuration



A backup of the LoadMaster configuration can be taken by going to **System Administration > Backup/ Restore** and clicking **Create Backup File**.

The configuration can be restored from this screen also. Please keep in mind that the Virtual Service settings can be restored by selecting **VS Configuration** and the rules can be restored by selecting **LoadMaster Base Configuration**.

Note: A WAF configuration can only be restored onto a LoadMaster with a WAF license.

6

References

Other useful documents are as follows:

ModSecurity Reference Manual

Web Application Firewall Feature Description

LoadMaster Product Overview