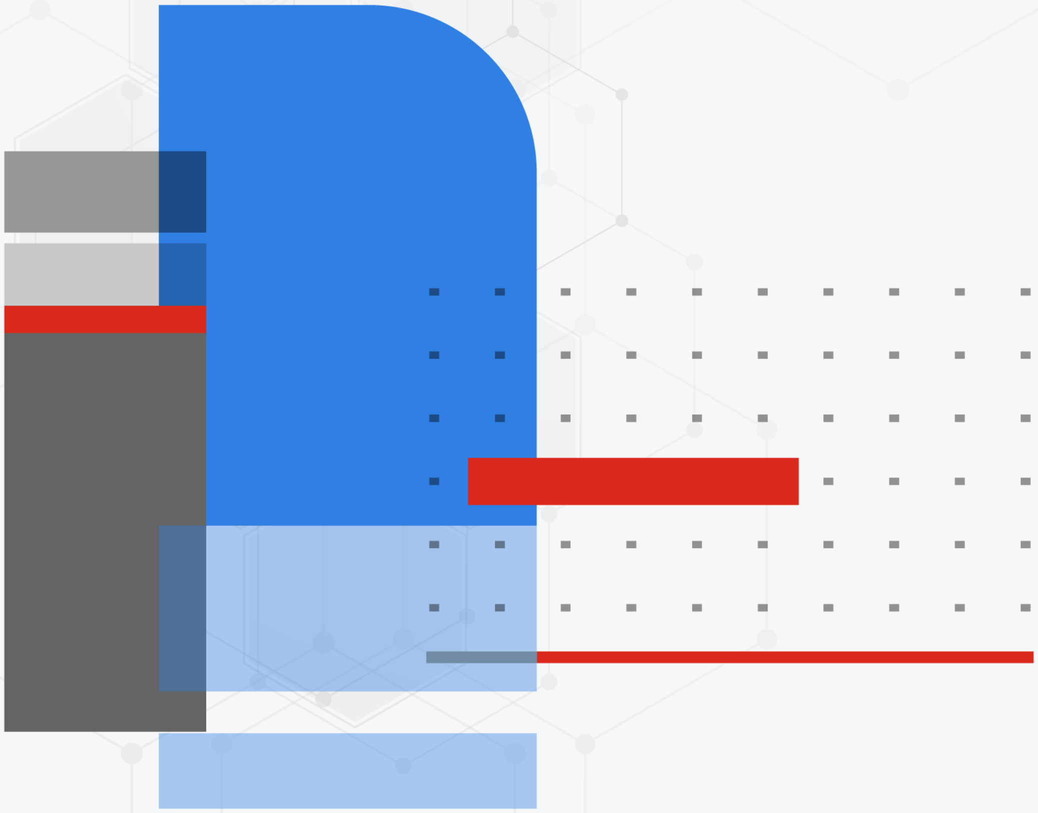
Script Reference Guide

FortiADC 8.0.0



FORTINET®

December 5, 2025
FortiADC Script Reference Guide

TABLE OF CONTENTS

Change Log

Date	Change Description
December 5, 2025	Initial release.

Overview

Lua scripting in FortiADC allows you to customize the behavior of your application delivery controller using Lua scripts. FortiADC supports Lua scripting primarily for defining custom content rules, manipulating HTTP requests and responses (through HTTP Scripting), manipulating TCP or UDP requests and responses (through Stream Scripting), and performing advanced traffic handling tasks.

The following outlines the basic overview of how Lua scripting works in FortiADC:

Integration with FortiADC

Lua scripting is integrated into the FortiADC platform, allowing you to write Lua scripts that can be executed at various stages of the request-response cycle.

Event Triggers

Lua scripts can be triggered at different stages of the HTTP, TCP, or UDP request-response cycle. This includes events such as when a request is received, before forwarding a request to the backend server, or after receiving a response from the server.

Access to Request and Response Data

Lua scripts have access to the request and response data, for HTTP Scripting, this includes headers, cookies, and payloads. This allows you to inspect and modify the request and response as needed.

Custom Logic

You can write custom logic in Lua scripts to implement specific functionality based on your requirements. This could include routing decisions, content manipulation, header enrichment, authentication, logging, and more.

Integration with FortiADC Features

Lua scripting can be used to extend the capabilities of FortiADC by integrating with its features such as load balancing, SSL offloading, caching, and traffic management policies.

HTTP Scripting and Stream Scripting

FortiADC supports two types of Lua scripting implementation: HTTP Scripting and Stream Scripting. HTTP Scripting is supported in Layer 2 and Layer 7 HTTP/HTTPS virtual servers, whereas Stream Scripting is support in Layer 7 TCP or UDP virtual servers.

For details, please refer to the respective sections for HTTP Scripting on page 19 and Stream Scripting on page 381.

Key concepts and features

This section covers key concepts and features you can reference when implementing Lua scripting in FortiADC:

- Definition of terms on page 7
- Control structures on page 8
- Operators on page 9
- Functions on page 13
- String library on page 15
- Special characters on page 16

Definition of terms

Frontend

When visiting the virtual server service, the client will create a connection with FortiADC – this connection between the client and the virtual server is referred to as the "frontend".

Backend

After FortiADC receives the request from the client, FortiADC will load-balance the request back to the real servers – this connection between FortiADC and the real server is referred to as the "backend".

Source IP address (frontend)

From the frontend connection, the source IP address refers to the **client address**.

Destination address (frontend)

From the frontend connection, the destination address refers to the **virtual server address**.

Source IP address (backend)

From the backend connection, the source IP address refers to **FortiADC's outgoing interface IP address**.

Destination address (backend)

From the backend connection, the destination IP address refers to the **real server address**.

Lua comment

All Lua comment lines must be prefixed with --; block comments (--[[...]]) are not yet supported.

Control structures

The table below lists Lua control structures.

Type	Structure
if then else	if condition1 then … elseif condition2 then … break else … go to location1 end ::location1::
for	--fetch all values of table 't' for k, v in pairs(t) do … end

Operators

Operators are symbols used to perform specific mathematical or logical manipulations.

The Lua language includes the following categories of built-in operators:

- Arithmetic Operators on page 9
- Bitwise Operators on page 9
- Relational Operators on page 11
- Logical Operators on page 11
- Misc Operators on page 11

For more details, see the Lua Reference Manual.

Arithmetic Operators

The Arithmetic operators listed below all operate on real numbers.
In the **Example** column, assume variable **A** is 10 and variable **B** is 20.

Operator	Description	Example
+	Adds two operands.	A + B results in 30.
-	Subtracts second operand from the first.	A - B results in -10.
*	Multiplies both operands.	A * B results in 200.
/	Divides the numerator by the denominator.	B / A results in 2.
//	Floor division is a division that rounds the quotient towards minus infinity, that is, the floor of the division of its operands. This rounds the result down to the nearest whole number or integer, which is less than or equal to the normal division result.	B // A results in 0.
%	Modulo is the remainder of an integer division that rounds the quotient towards minus infinity (floor division).	B % A results in 0.
^	Exponentiation.	A^2 results in 100.
-	Unary minus acts as negation.	-A results in -10.

Bitwise Operators

Bitwise operators convert its operands to integers, operate on all bits of those integers, and result in an integer.
In the **Example** column, assume we have two integer variables with binary representations:

```
a = 1100; // (12 in decimal)
b = 1010; // (10 in decimal)
```

Operator	Description	Example

&	Bitwise AND compares each bit of the first operand with the corresponding bit of the second operand. If both bits are 1, the corresponding result bit is set to 1. Otherwise, the result bit is set to 0.	`c = a & b; // (8 in decimal)`
		`c = 1000;`

|	Bitwise OR compares each bit of the first operand to the corresponding bit of the second operand. If either of the bits is 1, the corresponding result bit is set to 1. Otherwise, the result bit is set to 0.	`c = a | b; // (14 in decimal)`
		`c = 1110;`

~	Bitwise exclusive OR (XOR) compares each bit of the first operand to the corresponding bit of the second operand. If the bits are different, the corresponding result bit is set to 1. Otherwise, the result bit is set to 0.	`c = a ~ b; // (6 in decimal)`
		`c = 0110;`

>>	Bitwise Right Shift shifts the bits of the given number to the right by the specified positions. When a number is right shifted, 0s are added to the left side of the number, and the rightmost bits are discarded.	Using a variable "n" and a shift count "s", the left shift operation can be represented as:
		`c = n >> s;`
		`a = 1100; (12 in decimal)`
		If we right shift "a" by 2 positions, the resulting value is:
		`c = a >> 2; // (3 in decimal)`
		`c = 11;`

<<	Bitwise Left Shift shifts the bits of the given number to the left by the specified positions. When a number is left shifted, 0s are added to the right side of the number. The leftmost bits are discarded.	Using a variable "n" and a shift count "s", the left shift operation can be represented as:
		`c = n << s;`
		`a = 1010; (10 in decimal)`
		If we left shift "a" by 2 positions, the resulting value is:
		`c = a << 2; // (40 in decimal)`
		`c = 101000;`

~	Unary bitwise NOT is used to perform bitwise negation on a single operand. It flips all the bits of the operand, changing each - bit to 1 and each 1 bit to 0.	Using the variable "a":
		`a = 1010; (10 in decimal)`
		`c = ~a`
		`c = 11111111111111111111111111110101; (-11 in decimal)`
		Applying the unary bitwise NOT operator to "10" flips all the bits, resulting in the Lua integer "-11".

Relational Operators

Lua provides the Relational operators listed below. This type of operator always results in **true** or **false**.
In the **Example** column, assume variable **A** is 10 and variable **B** is 20.

Operator	Description	Example
==	Checks if the value of two operands are equal or not. If yes, then the condition is true.	(A == B) is false.
~=	Checks if the value of two operands are equal or not. If the values are not equal, then the condition is true.	(A ~= B) is true.
>	Checks if the value of the left operand is greater than the value of the right operand. If yes, then the condition is true.	(A > B) is not false.
<	Checks if the value of the left operand is less than the value of the right operand. If yes, then the condition is true.	(A < B) is true.
>=	Checks if the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition is true.	(A >= B) is false.
<=	Checks if the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition is true.	(A <= B) is true.

Logical Operators

Lua provides the Logical operators listed below. This type of operator considers false and nil both as **false**, and anything else as **true**.

In the **Example** column, assume variable **A** is 10 and variable **B** is 20.

Operator	Description	Example
and	Performs a logical "and" comparison between two values. If both the operands are non-zero then the condition is true. The operator **and** returns its first argument if it is false; otherwise, it returns its second argument.	(A and B) is 20.
or	Performs a logical "or" comparison between two values. If any of the two operands is non-zero, then the condition is true. The operator **or** returns its first argument if it is not false; otherwise, it returns its second argument	(A or B) is 10.
not	Performs a logical "not" on a value. Used to reverse the logical state of its operand. If a condition is true, then the Logical **not** operator will make it false.	not(A and B) is false.

Misc Operators

Miscellaneous operators supported by Lua include **concatenation** and **length**.

Operator	Description	Example
..	The string concatenation operator in Lua is denoted by two dots ('..'). If both operands are strings or numbers, then they are converted to a string. It's the same as __concat.	a..b where a is "Hello " and b is "World", will return "Hello World".

Functions

FortiADC supports the basic commands. If the user wants more functions, the user can implement functions to define more with the basic commands.

FortiADC supports the

Syntax

```
function function_name(parameter)
…
end
```

Examples: cookie command usage

FortiADC supports two cookie commands: cookie_list() and cookie(t) with t as a table input

```
when HTTP_REQUEST {
ret=HTTP:cookie_list()
for k, v in pairs(ret) do
debug("-----cookie name %s, value %s-----\n", k, v);
end
```

GET value of cookie "test"

```
value = get_cookie_value(HTTP, "test") --the return value is either boolean false or its value if
      exists.
debug("-----get cookie value return %s-----\n", value);
```

GET attribute path of cookie "test", can be used to get other attributes too

```
case_flag = 0; -- or 1
ret = get_cookie_attribute(HTTP, "test", "path", case_flag);--return value is either boolean false
      or its value if exists
debug("-----get cookie path return %s-----\n", ret);
```

SET value of cookie "test"

```
ret = set_cookie_value(HTTP, "test", "newvalue")--return value is boolean
debug("-----set cookie value return %s-----\n", ret);
```

REMOVE a whole cookie

```
ret = remove_whole_cookie(HTTP, "test")--return value is boolean
debug("-----remove cookie return %s-----\n", ret);
```

INSERT a new cookie.

You need to make sure the cookie was not first created by the GET command. Otherwise, by design FortiADC shall use SET command to change its value or attributes.

In HTTP REQUEST, use $Path, $Domain, $Port, $Version; in HTTP RESPONSE, use Path, Domain, Port, Version, etc.

```
ret = insert_cookie(HTTP, "test", "abc.d; $Path=/; $Domain=www.example.com, ")--return value is
      boolean
debug("-----insert cookie return %s-----\n", ret);
}
function get_cookie_value(HTTP, cookiename)
local t={};
t["name"]=cookiename
t["parameter"]="value";
t["action"]="get"
return HTTP:cookie(t)
end
```

attrname can be path, domain, expires, secure, maxage, max-age, httponly, version, port.

case_flag: If you use zero, FortiADC looks for default attributes named Path, Domain, Expires, Secure, Max-Age, HttpOnly, Version, Port. By setting this to 1, you can specify the case sensitive attribute name to look for in t ["parameter"] which could be PAth, DOmaIn, MAX-AGE, EXpires, secuRE, HTTPONLy, VerSion, Port, etc.

```
function get_cookie_attribute(HTTP, cookiename, attrname, case_flag)
local t={};
t["name"]=cookiename
t["parameter"]=attrname;
t["case_sensitive"] = case_flag;
t["action"]="get"
return HTTP:cookie(t)
end
function set_cookie_value(HTTP, cookiename, newvalue)
local t={};
t["name"]=cookiename
t["value"]=newvalue
t["parameter"]="value";
t["action"]="set"
return HTTP:cookie(t)
end
function remove_whole_cookie(HTTP, cookiename)
local t={};
t["name"]=cookiename
t["parameter"]="cookie";
t["action"]="remove"
return HTTP:cookie(t)
end
function insert_cookie(HTTP, cookiename, value)
local t={};
t["name"]=cookiename
t["value"]=value;
t["parameter"]="cookie";
t["action"]="insert"
return HTTP:cookie(t)
end
```

String library

The FortiADC OS supports only the Lua string library. All other libraries are disabled. The string library includes the following string-manipulation functions:

- `string.byte(s, i)`
- `string.char(i1,i2…)`
- `string.dump(function)`
- `string.find(s, pattern)`
- `string.format`
- `string.gmatch`
- `string.gsub`
- `string.len`
- `string.lower`
- `string.match`
- `string.rep`
- `string.reverse`
- `string.sub`
- `string.upper`
- `string.starts_with`
- `string.ends_with`

Note:

- If you want to use regular expression match, you can use `string.match` with Lua patterns.
- All relational operators >, <, >=, <=, ~=, == apply to strings. Especially, == can be used to test if one string equals to another string.
- `string.find` can be used to test whether one string contains another string.

For a tutorial on scripting with the Lua string library, see http://lua-users.org/wiki/StringLibraryTutorial.

For example: `uri:starts_with (b), uri:ends_with (b)`

Example 1:

```
a=12

b=string.format("%s, pi=%.4f", a, 3.14)

• {s:byte(1,-1)}
```

Example 2:

```
str = "abc\1\2\0"
t={str:byte(1,-1)}
t is a table like an array to store the string.
```

```
t[1] is 97, t[2] is 98, t[3] is 99, t[4] is 1, t[5] is 2, t[6] is 0.
• {s:sub(1, 3)}
str="abcdefg"
```

Example 3:

```
str = “abc\1\2\0”
t={str:byte(1,-1)}
t is a table like an array to store the string.
t[1] is 97, t[2] is 98, t[3] is 99, t[4] is 1, t[5] is 2, t[6] is 0.

• {s:sub(1, 3)}

str="abcdefg"
test=str:sub(2,5)
debug("return: %s \n", test)
log("record a log")

result:   “bcde”
```

Special characters

This section discusses the use of special characters in FortiADC scripting.

Log and debug

FortiADC supports the special characters as listed in Special characters and ways to handle them on page 16 in log and debug scripts.

Special characters and ways to handle them

Character	Name
~	Tilde
!	Exclamation
@	At sign
#	Number sign (hash)
$	Dollar sign
^	Caret
&	Ampersand
*	Asterisk
(	Left parenthesis

Character	Name
)	Right parenthesis
_	Underscore
+	Plus
{	Left brace
}	Right brace
[	Left bracket
]	Right bracket
.	Full stop
?	Question mark

When written in a string, these characters look like this (between double quotes: "`~!@#$^&*()_+{}[].?`"

Note: The back slash (\) and the percent (%) signs are handled in a unique way in log and debug scripts. To print out %, you must use %%; to print out \, you must use \\.

HTTP data body commands

HTTP data body commands, such as `find, remove,` and `replace`support regular expression, which treats special characters such as (between double quote) "`$^?*+.|()[]{}\`" in a special way. You MUST escape these characters to demolish their special meaning. Special characters in HTTP data body commands on page 17 shows how to escape these special characters.

Special characters in HTTP data body commands

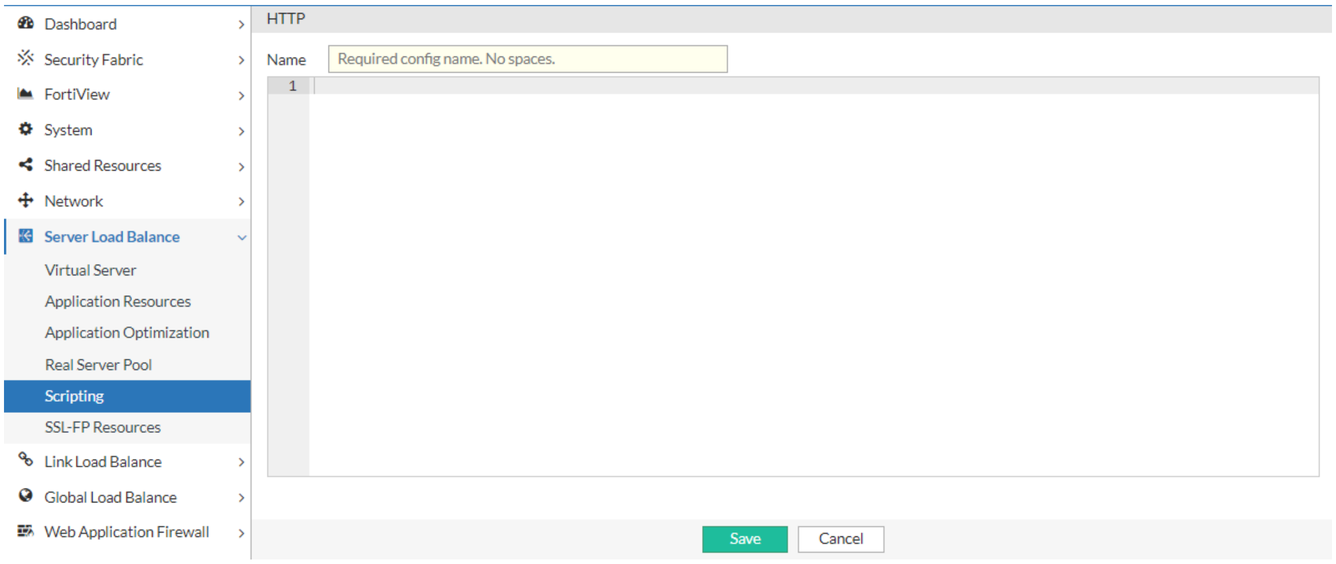
To print out ...	You MUST use ...
$	\\$
^	\\^
?	\\?
*	*
+	\\+
.	\\.
|	\\\|
\	\\\\
(and)	\\(and \\)
{and}	\\{and \\}
[and]	\\[and \\]

Note:

- { and } are special because the script syntax looks for the matching { and }. So be sure to use them in pairs.
- The `find, remove,` and `replace` commands use special expression. Particularly, `p.ge` will match the whole word page and remove and replace the whole word page. However, `p*ge` will remove and replace only the `ge` part.
- The HTTP data body `set` command does not support regular expression. Only \ is special in the `set`command, and you must use \\ for it.

HTTP Scripting

HTTP Scripting in FortiADC allows you to perform actions that are not supported by the current built-in feature set for Layer 2 and Layer 7 HTTP/HTTPS virtual servers. You can import HTTP scripts from the FortiADC GUI. To get started, FortiADC provides system predefined scripts that can be cloned for customization. The HTTP scripts are event-triggered, allowing you to manipulate HTTP requests and responses, redirection, and dynamically change backend routing. This functionality can be combined with other HTTP related functions such as WAF, SSL, and Authentication.



This section covers the following:

- HTTP Scripting configuration overview on page 19
- HTTP Scripting events on page 29
- Predefined HTTP Scripting commands on page 31
- Predefined HTTP scripts on page 359
- HTTP Scripting examples on page 368

HTTP Scripting configuration overview

You can use HTTP scripts to perform actions that are not currently supported by the built-in feature set for Layer 2 and Layer 7 HTTP/HTTPS virtual servers. Scripts enable you to use predefined script commands and variables to manipulate the HTTP request/response or select a content route, or get SSL information.

HTTP scripts are composed of several functional components that define the trigger events, commands, operators, and more. The following example demonstrates how HTTP scripting is applied to rewrite the HTTP Host header and path in an HTTP request.

The HTTP script:

```
when RULE_INIT {
debug("rewrite the HTTP Host header and path in a HTTP request \n")
}

when HTTP_REQUEST {
host = HTTP:header_get_value("Host")
path = HTTP:path_get()
if host:lower():find("myold.hostname.com") then
debug("found myold.hostname.com in Host %s \n", host)
HTTP:header_replace("Host", "mynew.hostname.com")
HTTP:path_set("/other.html")
end
}
```

HTTP script component breakdown:

Parameter	Example	Description
Events – for details, see HTTP Scripting events on page 29.		
	RULE_INIT	The event is used to initialize global or static variables used within a script. It is triggered when a script is added or modified, or when the device starts up, or when the software is restarted.
	HTTP_REQUEST	The virtual server receives a complete HTTP request header.
Commands – for details, see Predefined HTTP Scripting commands on page 31.		
	debug(str)	Prints the debug information when VS using scripting.
	HTTP:header_get_values (header_name)	Returns a list of value(s) of the HTTP header named <header_name>, with a count for each value. Note that the command returns all the values in the headers as a list if there are multiple headers with the same name.
	HTTP:path_get()	Returns the string of the HTTP request path.
	HTTP:header_replace(header_ name, value)	Replaces the occurrence value of header <header_ name> with value <value>.
	HTTP:path_set(value)	Sets HTTP request path to the string <value>.
Operators – for details, see Operators on page 9. (Not applicable in this example).		
Strings – for details, see String library on page 15.		
	host:lower():find ("myold.hostname.com")	The string library includes the string-manipulation functions, such as: `string.lower` `string.find(s, pattern)`

Parameter	Example	Description
		This example combines the above string manipulation functions, using `lower()` to convert the Host strings to lowercase and then `find()` to search for the Host header for a match.
Control structures – for details, see Control structures on page 8.		

	if...then
	end
		if condition1 then
		…
		else if condition2 then
		… break
		else
		… go to location1
		end
		
		::location1::

Functions – for details, see Functions on page 13. (Not applicable in this example).		

Configuring HTTP Scripting

From the FortiADC GUI, you can type or paste the script content into the configuration page. Alternatively, you can clone a system predefined script to customize. For details, see Predefined HTTP scripts on page 359. From the HTTP Script page, you also have the option to import, export, and delete scripts.

Before you begin:

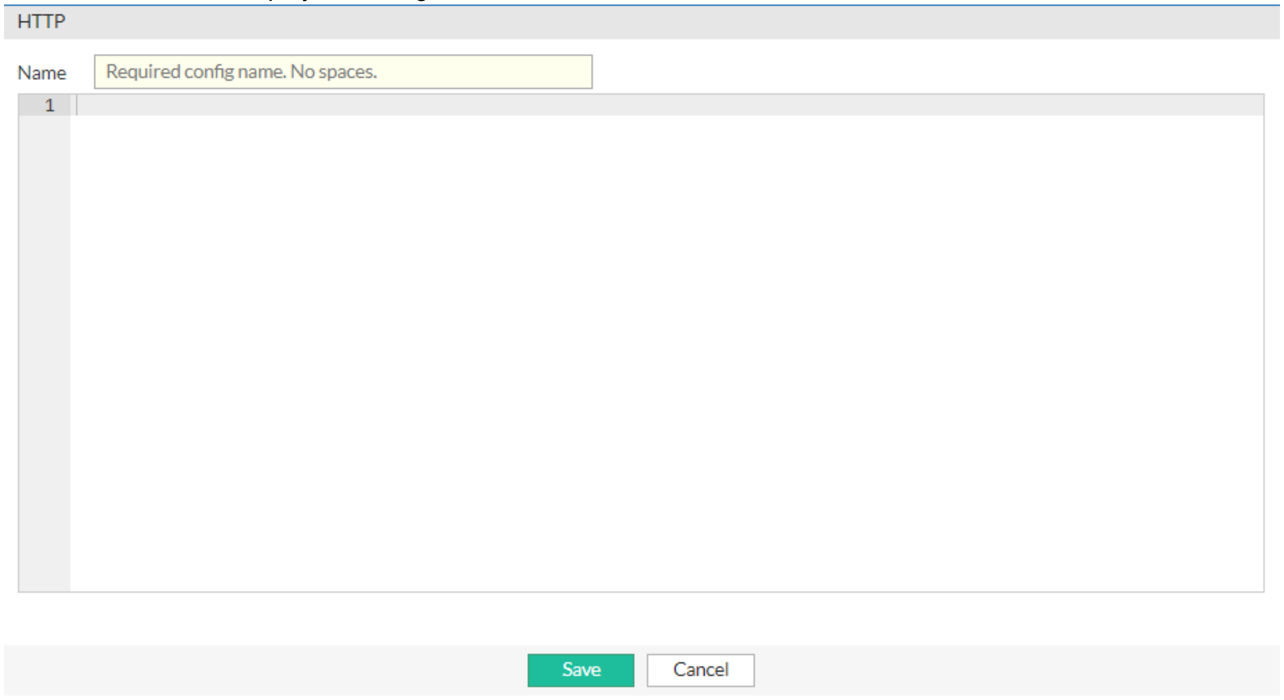
- You must have Read-Write permission for Server Load Balance settings.

After you have created a script configuration object, you can specify it in the virtual server configuration.

To create an HTTP script configuration object:

1. Go to **Server Load Balance > Scripting**.
   The configuration page displays the **HTTP** tab.

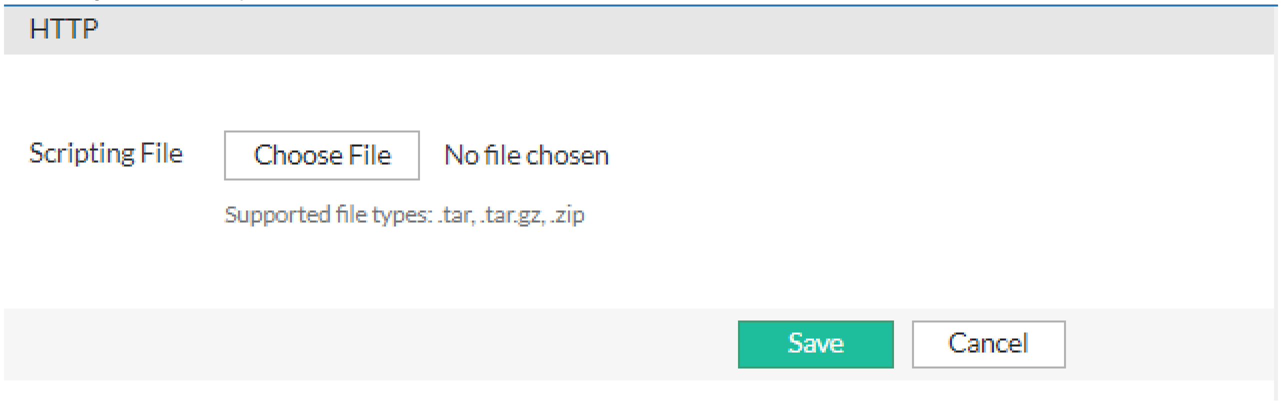
2. Click **Create New** to display the configuration editor.



3. Enter a unique name for the HTTP script configuration. Valid characters are `A-Z`, `a-z`, `0-9`, `_`, and `-`. No spaces. After you initially save the configuration, you cannot edit the name.
4. In the text box, type or paste your HTTP script.
   If you want to include this script as part of a multi-script configurations that allows you to execute multiple scripts in a certain order, ensure to set its priority. For more information, see Multi-script support for HTTP Scripting on page 25.
5. Click **Save**.
   Once the HTTP script configuration is saved, you can specify it in the virtual server.

To import an HTTP script:

1. Go to **Server Load Balance > Scripting**.
   The configuration page displays the **HTTP** tab.
2. Click **Import** to display the file import options.

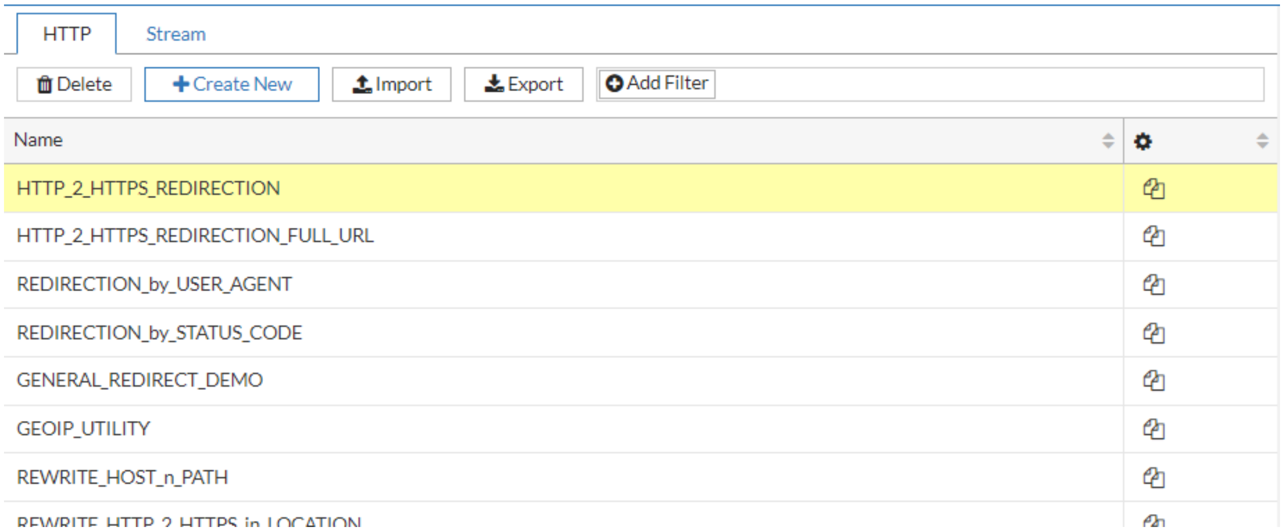


3. Click **Choose File** and browse for the script file. Supported file types are .tar, .tar.gz, and .zip.

4. Click **Save**.
   Once the file is successfully imported, it will be listed in the **Scripting > HTTP** page.

To export an HTTP script:

1. Go to **Server Load Balance > Scripting**.
   The configuration page displays the **HTTP** tab.
2. From the **HTTP** page, select an HTTP script configuration.
   In the example below, the HTTP_2_HTTPS_REDIRECTION script is selected.

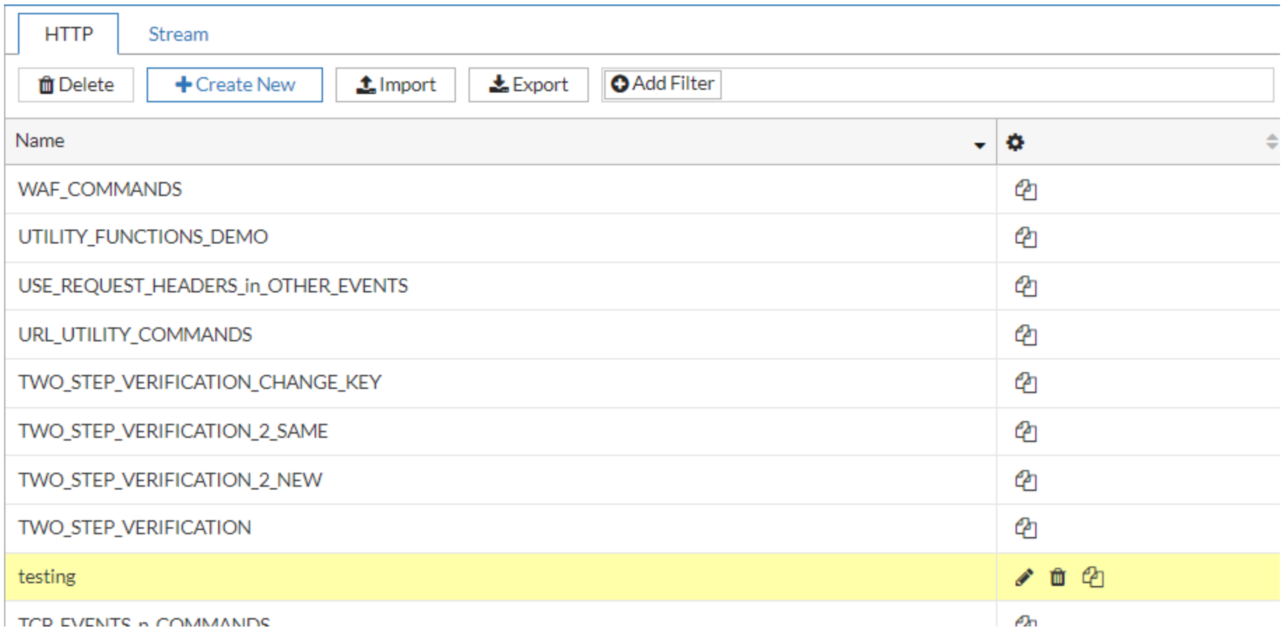


Name	
HTTP_2_HTTPS_REDIRECTION	
HTTP_2_HTTPS_REDIRECTION_FULL_URL	
REDIRECTION_by_USER_AGENT	
REDIRECTION_by_STATUS_CODE	
GENERAL_REDIRECT_DEMO	
GEOIP_UTILITY	
REWRITE_HOST_n_PATH	
REWRITE_HTTP_2_HTTPS_in_LOCATION	

3. Click **Export** initiate the file download.
   The selected script configuration will be exported as a .tar file.

To delete an HTTP script:

1. Go to **Server Load Balance > Scripting**.
   The configuration page displays the **HTTP** tab.
2. From the **HTTP** page, select a user-defined HTTP script configuration. System predefined scripts cannot be deleted.
   In the example below, the **testing** script configuration is selected.



Name
WAF_COMMANDS
UTILITY_FUNCTIONS_DEMO
USE_REQUEST_HEADERS_in_OTHER_EVENTS
URL_UTILITY_COMMANDS
TWO_STEP_VERIFICATION_CHANGE_KEY
TWO_STEP_VERIFICATION_2_SAME
TWO_STEP_VERIFICATION_2_NEW
TWO_STEP_VERIFICATION
testing

3. Click **Delete** from the top navigation, or click (delete icon) of the configuration.
   Multiple script configurations can be deleted using the **Delete** button on the top navigation.

Multi-script support for HTTP Scripting

Linking multiple scripts to the same virtual server

FortiADC supports the use of a single script file containing multiple scripts and applies them to a single virtual server in one execution. Different scripts can contain the same event. You can specify the priority for each event in each script file to control the sequence in which multiple scripts are executed or allow the system to execute the individual scripts in the order they are presented in the multi-script file.

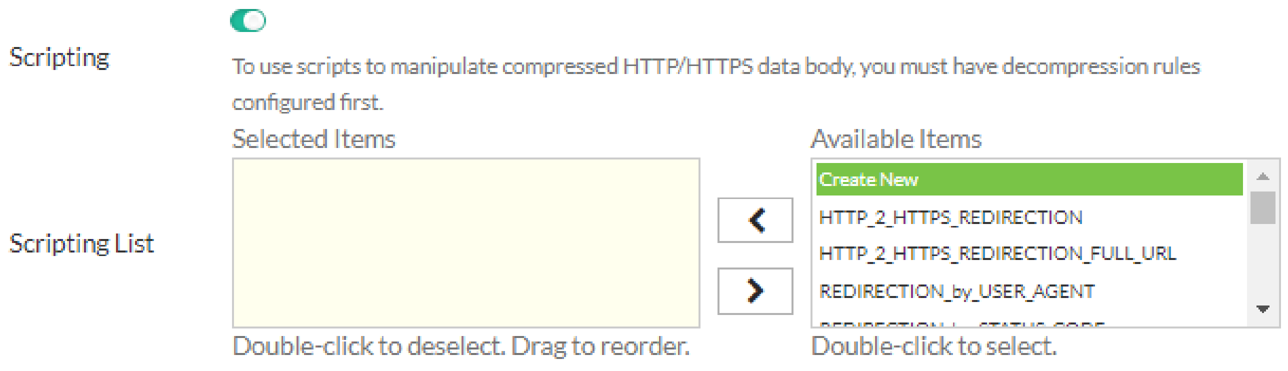
Currently, up to 16 individual scripts can be added to create a large multi-script file.

If desired, you can disable the processing of remaining scripts in the multi-script, or you can even complete disable the processing of certain events (for example, you can disable the processing of the HTTP RESPONSE event in a HTTP REQUEST script). FortiADC also supports multiple calls of HTTP:redirect(), HTTP:redirect_with_cookie(), LB:routing(), and HTTP:close() functions such that the final call prevails.

In practice, instead of creating a single large and complex script containing all necessary logic, it's often more advantageous to decompose it into smaller functional components represented by individual scripts. This approach offers several benefits. Firstly, executing multiple scripts concurrently is more efficient than running them sequentially. Additionally, breaking down a massive script into smaller units enhances flexibility, particularly when applying scripts to various virtual servers. Some servers may require only specific scripts, while others may utilize all available ones. With smaller, modular scripts, you have the flexibility to select and combine only the necessary components to construct a comprehensive multi-script file, each with its designated priority, and apply them collectively to a virtual server.

Apply multiple scripts on page 25 shows how to link multiple scripts to a single virtual server from the GUI.

Apply multiple scripts


Scripting
To use scripts to manipulate compressed HTTP/HTTPS data body, you must have decompression rules configured first.
Selected Items
Available Items
Scripting List
Create New
HTTP_2_HTTPS_REDIRECTION
HTTP_2_HTTPS_REDIRECTION_FULL_URL
REDIRECTION_by_USER_AGENT
Double-click to deselect. Drag to reorder.
Double-click to select.

Setting script priority

Priority in a multi-script is *optional*, but is highly recommended. When executing a big multiple-script file, care must be taken to avoid conflicting commands among the scripts. You can set the priority for each script using the script editor on FortiADC's GUI. Valid values range from 1 to 1,000, with 500 being the default. The smaller the value, the higher the priority. Below is an example script with a set priority:

```
when HTTP_REQUEST priority 100 {
LB:routing(“cr1”)
}
```

To display the priority information in the GUI, you can define one and only one event in each script file, as shown below:

Script 1:

```
when HTTP_REQUEST priority 500 {
LB:routing("cr1")
}
```

Script 2:

```
when HTTP_RESPONSE priority 500 {
HTTP:close()
}
```

Script 3:

```
when HTTP_REQUEST priority 400 {
LB:routing("cr2")
}
```

Script 4:

```
when HTTP_RESPONSE priority 600 {
HTTP:close()
}
```

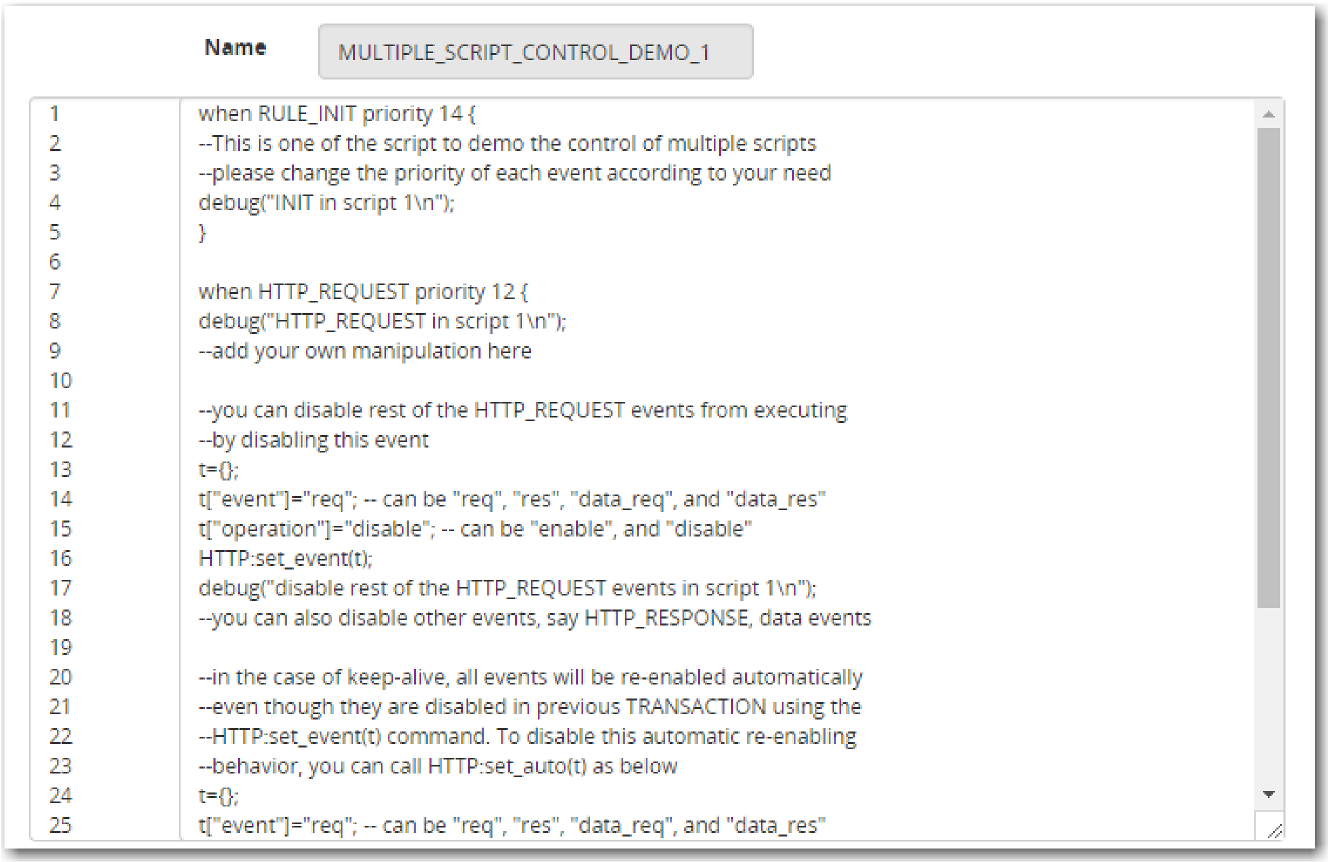
Individual script files are loaded separately into the Lua stack. A numeric value (starting from 1) is appended to each event (e.g., for HTTP_REQUEST event, there are functions `HTTP_REQUEST1`, `HTTP_REQUEST2,` and so on so forth).

To support multiple scripts, FortiADC:

- Supports multiple calls of redirect/routing/close function, making them re-entrant so that the final one prevails. For that purpose, the system checks the behavior of multiple calls across `redirect(), close(), and routing()`. If `redirect()` comes first, followed by `close(),` then `close()` prevails. If `close()` comes first, followed by `redirect()`, then `redirect()` prevails. If you want to `close()`, you must disable the event after `close()`.
- Allows enabling or disabling events. There are times when you may want to disable the processing of the remaining scripts while a multi-script file is being executed, or want to disable processing the response completely. The mechanism serves that purpose.
- Allows enabling or disabling automatic event-enabling behavior. In the HTTP keep-alive mode, the system by default re-enables HTTP REQUEST and HTTP RESPONSE processing for the next transaction (even if they are disabled in the current transaction using the above enable or disable event mechanism). Now you can disable or enable this automatic enabling behavior.

Script priority on page 26 shows a sample multi-script with priority information.

Script priority

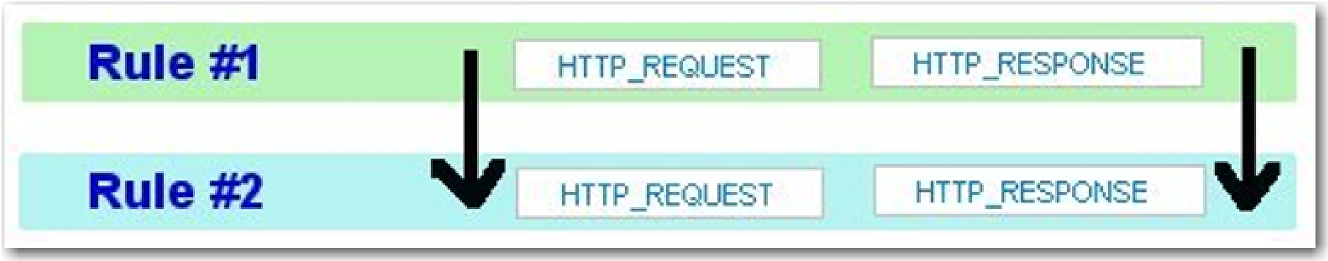


Name MULTIPLE_SCRIPT_CONTROL_DEMO_1

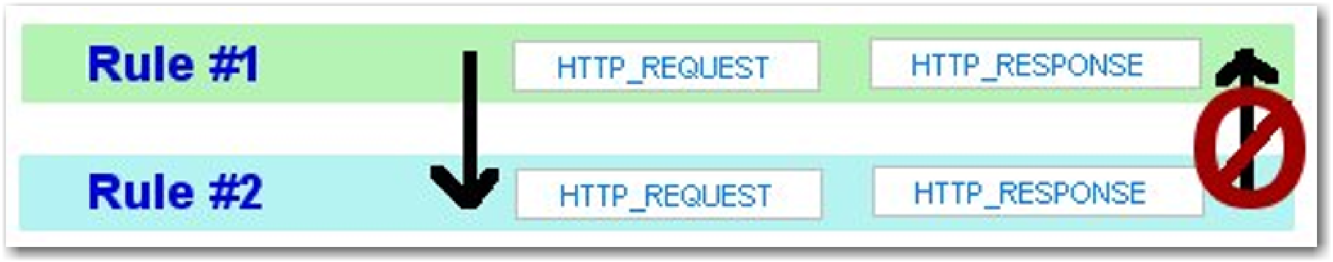
```
when RULE_INIT priority 14 {
--This is one of the script to demo the control of multiple scripts
--please change the priority of each event according to your need
debug("INIT in script 1\n");
}

when HTTP_REQUEST priority 12 {
debug("HTTP_REQUEST in script 1\n");
--add your own manipulation here

--you can disable rest of the HTTP_REQUEST events from executing
--by disabling this event
t={};
t["event"]="req"; -- can be "req", "res", "data_req", and "data_res"
t["operation"]="disable"; -- can be "enable", and "disable"
HTTP:set_event(t);
debug("disable rest of the HTTP_REQUEST events in script 1\n");
--you can also disable other events, say HTTP_RESPONSE, data events

--in the case of keep-alive, all events will be re-enabled automatically
--even though they are disabled in previous TRANSACTION using the
--HTTP:set_event(t) command. To disable this automatic re-enabling
--behavior, you can call HTTP:set_auto(t) as below
t={};
t["event"]="req"; -- can be "req", "res", "data_req", and "data_res"
```

Compiling principles

- All individual scripts should be pre-compiled when they are linked to a virtual server, where they can be combined into one big multi-script.
- For the same event, combine the commands in different scripts according to their priorities and orders.
- For commands of different priorities, FortiADC processes the high-priority commands first, and then the low-priority ones; for commands of the same priority, it processes them in the order they appear in the combined script.
- And if you are using multiple scripts with overlapping events for bidirectional traffic, you must ensure that the response traffic traverses the overlapping events in the expected order. By default, the scripts applied to the same virtual server will run in the order in which they are applied, regardless of the direction of traffic flow.
- For a specified event, you must make sure to avoid the conflict commands in different scripts. For example, if you have multiple scripts applied to the same virtual server and the scripts contain both request and response logic, the default execution order is like this:



but NOT like this:



As shown above, FortiADC cannot control the order in which events in the scripts are executed. The only way to enforce the execution order for response traffic is to use the event priority command, as we have discussed above. When setting the priorities, pay special attention to both request and response flows.

Special notes

When using the multi-script feature, keep the following in mind:

- The multi-script feature is supported on all FortiADC hardware platforms.
- Currently, the feature can be applied to Layer 2 and Layer 7 virtual servers on HTTP/HTTPS protocol only.
- Scripts are VDOM-specific, and cannot be shared among different VDOMs.
- Session tables set up using scripts must be synced through high-availability (HA) configuration.
- Each multi-script configuration can contain up to 256 individual scripts, each being no more than 32 kilobytes.

Dynamic scripting configuration change

When changing an in-use script, FortiADC supports dynamic reloading of the new scripting configuration.

HTTP Scripting events

HTTP Scripts are associated with a particular virtual server, and they are event-driven. A script is triggered when the associated virtual server receives an HTTP request or response. Then, it performs the programmed action.

Event	Description
HTTP_REQUEST	The virtual server receives a complete HTTP request header.
HTTP_RESPONSE	The virtual server receives a complete HTTP response header.
RULE_INIT	The event is used to initialize global or static variables used within a script. It is triggered when a script is added or modified, or when the device starts up, or when the software is restarted.
VS_LISTENER_BIND	The virtual server tries to bind.
SERVER_BEFORE_CONNECT	The virtual server is going to connect to the backend real server.
SERVER_CONNECTED	The HTTP proxy deems that the backend real server is connected.
AUTH_RESULT	The authentication (HTML Form / HTTP-basic) is done.
HTTP_DATA_REQUEST	Triggered whenever an HTTP:collect command finishes processing, after collecting the requested amount of data.
HTTP_DATA_RESPONSE	Triggered when an HTTP:collect command finishes processing on the server side of a connection.
CLIENTSSL_HANDSHAKE	The virtual server receives a complete HTTPS handshake on the client side.
SERVERSSL_HANDSHAKE	FortiADC receives a complete HTTPS handshake on the server side.
CLIENTSSL_RENEGOTIATE	The virtual server receives a re-connection request from a peer.
SERVERSSL_RENEGOTIATE	FortiADC sends a re-connection request to a peer.
TCP_ACCEPTED	The virtual server receives a complete TCP connection.
TCP_CLOSED	The virtual server close a TCP connection.
PERSISTENCE	Event hook inside process_sticking_rules() in httproxy.
POST_PERSIST	Event hook after LB is done and assigns real server according to ADC method.
SERVER_CLOSED	When Httproxy is going to terminate the backend real server connection.

Event	Description
COOKIE_BAKE	When FortiADC is done baking an authentication cookie. Allows PROXY commands, MGM commands and AUTH:get_baked_cookie/set_baked_cookie. The COOKIE_BAKE event can occur after AUTH_RESULT, HTTP_REQUEST, or HTTP_RESPONSE events.
BEFORE_AUTH	The BEFORE_AUTH event triggers right before the authentication is performed to allow the user specified user group to be used instead. The new user group will override the authentication result of the original authentication policy. HTTP: header_get_names header_get_values header_get_value header_remove header_remove2 header_insert header_replace header_replace2 header_exists header_count version_get version_set redirect_with_cookie redirect_t redirect close disable_event enable_event set_event set_auto disable_auto enable_auto rand_id get_session_id cookie cookie_list cookie_crypto respond method_get method_set uri_get uri_set path_get path_set query_get query_set client_port local_port remote_port client_addr local_addr remote_addr client_ip_ver LB: routing get_valid_routing get_current_routing method_assign_server AUTH: set_usergroup realm usergroup host SSL: renegotiate cert_request get_verify_depth set_verify_depth client_cert peer_cert cert IP: client_port local_port remote_port client_addr local_addr remote_addr client_ip_ver MGM: rand_id get_session_id disable_event enable_event set_event set_auto disable_auto enable_auto

WAF events

Use the WAF events to insert an action before or after a WAF scan.

In FortiADC, the WAF has six stages for when modules can scan for attacks:

- WAF_SCAN_STAGE_REQ_HEADER
- WAF_SCAN_STAGE_REQ_BODY (streaming stage)
- WAF_SCAN_STAGE_REQ_WHOLE_BODY
- WAF_SCAN_STAGE_RES_HEADER
- WAF_SCAN_STAGE_RES_BODY (streaming stage)
- WAF_SCAN_STAGE_RES_WHOLE_BODY

The WAF event may be applied to specific WAF stages depending on their hook point.

Event	Hook point	Example
WAF_REQUEST_BEFORE_SCAN	Before WAF_SCAN_STAGE_REQ_HEADER start.	when WAF_REQUEST_BEFORE_SCAN {

Event	Hook point	Example
	If WAF function is not enabled on VS, then this will not be triggered.	debug("test WAF_REQUEST_ BEFORE_SCAN\n") }
WAF_RESPONSE_BEFORE_ SCAN	Before WAF_SCAN_STAGE_RES_ HEADER start. If WAF function is not enabled on VS, then this will not be triggered.	when WAF_RESPONSE_BEFORE_ SCAN { debug("test WAF_RESPONSE_ BEFORE_SCAN\n") }
WAF_REQUEST_ATTACK_ DETECTED	After all request stages when there are attacks detected (violation). If WAF function is not enabled on VS, then this will not be triggered. If WAF module does not detect any violations, then this will not be triggered.	when WAF_REQUEST_ATTACK_ DETECTED { debug("test WAF_REQUEST_ ATTACK_DETECTED\n") }
WAF_RESPONSE_ATTACK_ DETECTED	After all response stages when there are attacks detected (violation). If WAF function is not enabled on VS, then this will not be triggered. If WAF module does not detect any violations, then this will not be triggered.	when WAF_RESPONSE_ATTACK_ DETECTED { debug("test WAF_RESPONSE_ ATTACK_DETECTED\n") }

Predefined HTTP Scripting commands

- Management commands on page 103
- IP commands on page 116
- TCP commands on page 130
- HTTP commands on page 204
- HTTP DATA commands on page 274
- HTTP Cookie commands on page 282
- Authentication commands on page 172
- SSL commands on page 149
- GEO IP commands on page 100
- HTTP RAM cache commands on page 288
- PROXY commands on page 319
- LB commands on page 312
- HTTP Persistence commands on page 301
- Global commands on page 32
- WAF commands on page 342

Global commands

- crc32(str) on page 35 – Computes the CRC-32 checksum for the input string and returns it as an unsigned integer.
- key_gen(str_pass, str_salt, iter_num, len_num) on page 36 – Creates an AES key for encrypt/decrypt data to use. Returns the generated key as a binary string.
- aes_enc(t) on page 37 – Encrypts the data using the previously-created AES key. Returns the encrypted data as a binary string.
- aes_dec(t) on page 39 – Decrypts the data using the previously-created AES key. Returns the decrypted data as a binary string.
- EVP_Digest(alg, str) on page 41 – Performs one-shot digest calculation using the specified algorithm.
- HMAC(alg, str, key) on page 42 – HMAC message authentication code.
- HMAC_verify(alg, data, key, verify) on page 43 – Compares the provided signature against the current digest value. Returns true for a match, false for a mismatch.
- G2F(alg, key) on page 44 – Returns a G2F random value.
- class_match(input_string, operation, pattern_table) on page 45 – Compares a string against a list of patterns or values using specified matching methods and returns detailed match information.
- class_search(pattern_table, operation, input_string) on page 47 – Searches through a list of strings to find elements that match the given pattern or criteria using specified matching methods.
- cmp_addr(client_ip, addr_group) on page 48 – Matches one IP address against a group of IP addresses. It can automatically detect IPv4 and IPv6 and can be used to compare IPv4 addresses with IPv6 addresses.
- url_enc(str) on page 50 – Converts the URL information into valid ASCII format.
- url_dec(str) on page 51 – Converts the encoding-URL into the original URL.
- url_parser(str) on page 52 – Parses a URL, returns a table containing host, port, path, query, fragment, the username, password, etc., from the URL.
- url_compare(url1, url2) on page 53 – Compares two URL strings, returns true if they are the same.
- rand() on page 54 – Generates a random number. Returns an integer number. After FortiADC reboots, the random number will be different.
- srand(str) on page 55 – Sets the random seed.
- rand_hex(int) on page 56 – Generates a random number in HEX. Returns a string, length is the <int>.
- rand_alphanum(int) on page 57 – Generates a random alphabet + number sequence. Returns a string, length is the <int>.
- rand_seq(int) on page 58 – Generates a random number sequence. Returns a string, length is the <int>.
- time() on page 59 – Returns the current time as a number in seconds. This is the time since the Epoch was measured.
- ctime() on page 60 – Returns the current time as a string, for example, "Tue Jun 25 14:11:01 2019".
- gmtime() on page 61 – Returns the GMT time as a string, for example, "Thu 27 Jun 2019 18:27:42 GMT".
- md5(str) on page 62 – Returns the MD5 calculated for the specified string.
- md5_hex(str) on page 63 – Returns the MD5 value in hex as a string.
- md5_str(str) on page 64 – Calculates the MD5 of a string input and stores the results in an intermediate variable, in some cases you need a version to deal with it.
- md5_hex_str(str) on page 65 – Calculates the MD5 of a string input of a string input and outputs the results in HEX format, in some cases you need a version to deal with it.
- sha1(str) on page 66 – Returns the SHA-1 calculated for the specified string.

- sha1_hex(str) on page 67 – Returns the SHA-1 calculated for the string in hex.
- sha1_str(str) on page 68 – Calculates the SHA-1 of a string input and stores the results in an intermediate variable, in some cases you need a version to deal with it.
- sha1_hex_str(str) on page 69 – Calculates the SHA-1 of a string input and output the results in HEX format, in some cases you need a version to deal with it.
- sha256(str) on page 70 – Calculates the SHA-256 of a string input and stores the result in an intermediate variable.
- sha256_hex(str) on page 71 – Calculates the SHA-256 of a string input and outputs the result in an intermediate variable. In some cases you need a version to deal with it.
- sha256_str(str) on page 72 – Calculates the SHA-256 of a string input and stores the result in an intermediate variable. In some cases you need a version to deal with it.
- sha256_hex_str(str) on page 73 – Calculates the SHA-256 of a string input and stores the result in an intermediate variable. In some case you need a version to deal with it.
- sha384(str) on page 74 – Calculates the SHA-384 of a string input and stores the result in an intermediate variable.
- sha384_hex(str) on page 75 – Calculates the SHA-384 of a string input and outputs the result in an intermediate variable. In some cases you need a version to deal with it.
- sha384_str(str) on page 76 – Calculates the SHA-384 of a string input and stores the result in an intermediate variable. In some cases you need a version to deal with it.
- sha384_hex_str(str) on page 77 – Calculates the SHA-384 of a string input and stores the result in an intermediate variable. In some case you need a version to deal with it.
- sha512(str) on page 78 – Calculates the SHA-512 of a string input and stores the result in an intermediate variable.
- sha512_hex(str) on page 79 – Calculates the SHA-512 of a string input and outputs the result in an intermediate variable. In some cases you need a version to deal with it.
- sha512_str(str) on page 80 – Calculates the SHA-512 of a string input and stores the result in an intermediate variable. In some cases you need a version to deal with it.
- sha512_hex_str(str) on page 81 – Calculates the SHA-512 of a string input and stores the result in an intermediate variable. In some case you need a version to deal with it.
- b32_enc(str) on page 82 – Encodes a string input in Base32 and outputs the result in string format.
- b32_enc_str(str) on page 83 – Encodes a string input in Base32 and outputs the result in string format. In some cases you need a version to deal with it.
- b32_dec(str) on page 84 – Decodes a Base32 encoded string input and outputs the result in string format.
- b32_dec_str(str) on page 85 – Decodes a Base32 encoded string input and outputs the result in string format. In some cases you need a version to deal with it.
- b64_enc(str) on page 86 – Encodes a string input in Base64 and outputs the result in string format.
- b64_dec(str) on page 87 – Decodes a Base64 encoded string input and outputs the result in string format.
- get_pid() on page 88 – Returns the PID value of the VS process.
- table_to_string(t) on page 89 – Returns the table in a string.
- htonl(int) on page 90 – Converts a 32 bit long integer from host byte order to network byte order.
- ntohs(int) on page 91 – Converts a 16 bit short integer from network byte order to host byte order.
- htons(int) on page 92 – Converts a 16 bit short integer from host byte order to network byte order.
- ntohl(int) on page 93 – When receiving long integers in HTTP response from the network, this command converts a 32 bit long integer from network byte order to host byte order.
- to_HEX(str) on page 94 – Returns the HEX calculate of the string.
- debug(str) on page 95 – Prints the debug information when the virtual server is using HTTP scripting.
- log(str) on page 96 – Prints the scripting running information in log format. When using this command, you should enable scripting log.

- file_open(path, str) on page 97 – Opens a file, returns a file object.
- file_gets(file, size) on page 98 – Returns the file content.
- file_close(file) on page 99 – Closes a file.

crc32(str)

Computes the CRC-32 checksum for the input string and returns it as an unsigned integer.

Syntax

crc32(str);

Arguments

Name	Description
str	Input string to calculate checksum for.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str = "any string for crc32 calculation"
crc = crc32(str);
debug("crc is %d\n", crc);
}
```

Supported Version

FortiADC version 5.2.x and later.

key_gen(str_pass, str_salt, iter_num, len_num)

Creates an AES key for encrypt/decrypt data to use. Returns the generated key as a binary string.

Syntax

key_gen(str_pass, str_salt, iter_num, len_num);

Arguments

Name	Description
str_pass	The password string.
str_salt	The salt string.
iter_num	The number of iterations.
len_num	The key length.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
new_key = key_gen("pass", "salt", 32, 32);     -- first parameter is the password string, second
the salt string, third the number of iterations, fourth the key length
debug("new key in hex is %s\n", to_HEX(new_key));
}
```

Supported Version

FortiADC version 5.2.x and later.

aes_enc(t)

Encrypts data using the AES algorithm with the specified key and key size.

Syntax

aes_enc(t);

Arguments

Name	Description
t	A table that specifies the message to encrypt, the encryption key, and the key size.

Events

Applicable in **all** events.

Operations

The function performs AES encryption on the provided message using the specified key.

Field	Type	Required	Description
message	String	Yes	The plaintext string to be encrypted.
key	String	Yes	The encryption key. Length must match the specified key size.
size	Integer	Yes	The AES key size. Must be 128, 192, or 256.

Key Length Requirements:

size=128: Key must be 16 bytes long

size=192: Key must be 24 bytes long

size=256: Key must be 32 bytes long

Example

```
when HTTP_REQUEST {
t={};
t["message"]  = "MICK-TEST";
t["key"]  = "aaaaaaaaaabbbbbb"            --16bit
t["size"]= 128        -- 128, 192, or 256, the corresponding key length is 16, 24, and 32
enc = aes_enc(t)
debug("The aes_enc output to HEX\n %s\n",to_HEX(enc));
}
```

Notes:

- Message: The plaintext string to be encrypted. (Note: Function will handle PKCS#7 padding internally).
- Key: A binary string representing the encryption key. Its length MUST be 16, 24, or 32 bytes for AES-128, AES-192, or AES-256 respectively. WARNING: Do not directly use a text password as the key. Use a Key Derivation Function (KDF) like PBKDF2 to create a secure key from a password.
- Size: The AES key size. Must be 128, 192, or 256.
- Output: The encrypted output is a binary string. Use the `to_HEX()` function to convert it to a hexadecimal representation for debugging or transmission.

Supported Version

FortiADC version 5.2.x and later.

aes_dec(t)

Decrypts data that was previously encrypted using the AES algorithm with the specified key and key size. Returns the decrypted data as a binary string.

Syntax

aes_dec(t);

Arguments

Name	Description
t	A table that specifies the encrypted data to decrypt, the decryption key, and the key size.

Operations

The function performs AES decryption on the provided encrypted data using the specified key.

Field	Type	Required	Description
message	String	Yes	The encrypted data as a binary string (direct output from aes_enc ()).
key	String	Yes	The decryption key. Length must match the key size (16, 24, or 32 bytes). Must be the same key used for encryption.
size	Number	Yes	The AES key size. Must be 128, 192, or 256.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
t={};
t["message"]  = "MICK-TEST";
t["key"]  = "aaaaaaaaaabbbbbb"
t["size"]= 128 -- 128, 192, or 256, the corresponding key length is 16, 24, and 32
enc = aes_enc(t)
--aes decryption
a={};
a["message"]  = enc;
a["key"]  = "aaaaaaaaaabbbbbb"
a["size"]= 128;
dec = aes_dec(a);
debug("key length %s decrypted is %s\n",”128” ,dec);
}
```

Notes:

- The message parameter for decryption should be the direct binary output from aes_enc() function.
- No hexadecimal conversion is needed between encryption and decryption.
- The key length must exactly match the AES key size requirement.
- Uses ECB mode with PKCS#7 padding.
- Automatically handles padding removal after decryption.

Supported Version

FortiADC version 5.2.x and later.

EVP_Digest(alg, str)

Performs one-shot digest calculation using the specified algorithm.

Syntax

EVP_Digest(alg, str);

Arguments

Name	Description
alg	A string to specify the type of hashing algorithms to use, must be MD5, SHA1, SHA256, SHA384, SHA512.
str	A string which will be EVP_Digested.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
alg = "MD5"; -- or "SHA1", "SHA256", "SHA384", "SHA512"
data = "your data"
re = EVP_Digest(alg, data);
debug("the digest in hex is %s\n", to_HEX(re));
}
```

Supported Version

FortiADC version 5.2.x and later.

HMAC(alg, str, key)

HMAC message authentication code.

Syntax

HMAC(alg, str, key);

Arguments

Name	Description
alg	A string which specifies the type of hashing algorithms to use, must be MD5, SHA1, SHA256, SHA384, SHA512.
str	A string which will be calculated.
key	A string which is a secret key.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
alg = "MD5"; -- or "SHA1", "SHA256", "SHA384", "SHA512"
data = "your data"
key  = "123456789ABCDEF0123456789ABCDEF\121"; -- or you can generate a key using key_gen
re = HMAC(alg, data, key);
debug("the HMAC in hex is %s\n", to_HEX(re));
}
```

Supported Version

FortiADC version 5.2.x and later.

HMAC_verify(alg, data, key, verify)

Compares the provided signature against the current digest value. Returns true for a match, false for a mismatch.

Syntax

HMAC_verify(alg, data, key,verify);

Arguments

Name	Description
alg	A string which specifies the type of hashing algorithms to use, must be MD5, SHA1, SHA256, SHA384, SHA512.
key	A string which is a secret key.
data	A string which will be calculated.
verify	A signature to compare the current digest against.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
alg = "MD5"; -- or "SHA1", "SHA256", "SHA384", "SHA512"
data = "your data"
verify = "your result to compare"
key  = "123456789ABCDEF0123456789ABCDEF\121"; -- or you can generate a key using key_gen
re = HMAC_verify(alg, data, key, verify);
if re then
debug("verified\n")
else
debug("not verified\n")
end
}
```

Supported Version

FortiADC version 5.2.x and later.

G2F(alg, key)

Returns a G2F random value.

Syntax

G2F(alg, key);

Arguments

Name	Description
alg	A string which specifies the algorithm.
key	A string which is a secret key.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
alg = "MD5"; -- or "SHA1", "SHA256", "SHA384", "SHA512"
key  = "123456789ABCDEF0123456789ABCDEF\121"; -- or you can generate a key using key_gen
re = G2F(alg, key);
debug("the G2F value is %d\n", re);
}
```

Note:

Alg: type of hashing algorithms to use, must be MD5, SHA1, SHA256, SHA384, SHA512

Supported Version

FortiADC version 5.2.x and later.

class_match(input_string, operation, pattern_table)

Check if input string satisfies operations with patterns.

Syntax

class_match(input_string, operation, pattern_table);

Arguments

Name	Description
input_string	The input string to match against the list.
operation	Operations that support "ends_with" "starts_with" "contains" "equals" operations.
pattern_table	A table that lists the strings to match against.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
    url_list = {"/admin", "/api", "/private", "/secure"}
    url = HTTP:uri_get()
    status, count, t = class_match(url, "starts_with", url_list)
    debug("status %s, count %s\n", status, count)
    -- Safe iteration with type checking
    if type(t) == "table" then
        for k, v in pairs(t) do
            debug("index %s, value %s\n", k, v)
        end
    else
        debug("ERROR: Expected table but got: %s\n", type(t))
    end
}
```

This command returns three parameters:

- **status**: a boolean indicating whether at least one match was found (true) or not (false).
- **count**: an integer representing how many matches were found in the list.
- **t**: a table containing the matched indexes and their corresponding values.

Supported Version

FortiADC version 5.2.x and later.

class_search(pattern_table, operation, input_string)

Searches through a list of strings to find elements that match the given pattern or criteria using specified matching methods.

Syntax

class_search(pattern_table, operation, input_string);

Arguments

Name	Description
pattern_table	A table that lists the strings to search through.
operation	Operations that support "ends_with" "starts_with" "contains" "equals" operations.
input_string	The pattern or string to match against the list elements.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
status, count, t = class_search(url_list, "starts_with", url);     --or "ends_with",  "equals",
"contains"
for k, v in pairs(t) do
debug("index %s, value %s\n", k, v);
end
}
```

This command returns three parameters:

- **status**: a boolean indicating whether at least one match was found (true) or not (false).
- **count**: an integer specifying how many elements in the list match the pattern.
- **t**: a table containing the matched list elements and their corresponding indexes.

Supported Version

FortiADC version 5.2.x and later.

cmp_addr(client_ip, addr_group)

Matches one IP address against a group of IP addresses. It can automatically detect IPv4 and IPv6 and can be used to compare IPv4 addresses with IPv6 addresses.

Syntax

cmp_addr(client_ip, addr_group);

Arguments

Name	Description
client_ip	For an IPv4 ip_addr/[mask], the mask can be a number between 0 and 32 or a dotted format like 255.255.255.0 For an IPv6 ip_addr/[mask], the mask can be a number between 0 and 128.

addr_group	A group of IP address.
	addr_group = "192.168.1.0/24" --first network address
	addr_group = addr_group..",::ffff:172.30.1.0/120" --second
	network address

Events

Applicable in **all** events.

Example

```
when RULE_INIT {--initialize the address group here
--for IPv4 address, mask can be a number between 0 to 32 or a dotted format
--support both IPv4 and IPv6, for IPv6, the mask is a number between 0 and 128
addr_group = "192.168.1.0/24"
addr_group = addr_group..",172.30.1.0/255.255.0.0"
addr_group = addr_group..",::ffff:172.40.1.0/120"
}

when HTTP_REQUEST {
client_ip = HTTP:client_addr()
matched = cmp_addr(client_ip, addr_group)
if matched then
debug("client ip found in address group\n");
else
debug("client ip not in address group\n");
end
}
```

Supported Version

FortiADC version 4.8.x and later.

url_enc(str)

Converts the URL information into valid ASCII format.

Syntax

url_enc(str);

Arguments

Name	Description
str	A string which will be converted.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
url_list ="https://abc.www.123.bbEEE.com/?5331=212&qe1=222"
debug("Ori= %s \nencodeed= %s\n", url_list,url_enc(url_list));
}
```

Supported Version

FortiADC version 5.2.x and later.

url_dec(str)

Converts the encoding-URL into the original URL.

Syntax

url_dec(str);

Arguments

Name	Description
str	A string which will be converted.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str = "http%3A%2F%2Fwww.example.com%3A890%2Furl%2Fpath%2Fdata%3Fname%3Dforest%23nose“
debug("String= %s\ndecoded= %s\n", str,url_dec(str));
}
```

Supported Version

FortiADC version 5.2.x and later.

url_parser(str)

Parses a URL, returns a table containing host, port, path, query, fragment, the username, password, etc., from the URL.

Syntax

url_parser(str);

Arguments

Name	Description
str	A url which will be parser.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
url_list="http://foo:bar@w1.superman.com/very/long/path.html?p1=v1&p2=v2#more-details"
purl = url_parser(url_list);
debug("parsed url scheme %s host %s\n port %s path %s query %s\n fragment %s, the username %s\n
passowrd %s\n", purl["scheme"], purl["host"], purl["port"],purl["path"], purl["query"], purl
["fragment"], purl["username"], purl["password"]);
}
```

Supported Version

FortiADC version 5.2.x and later.

url_compare(url1, url2)

Compares two URL strings, returns true if they are the same.

Syntax

url_compare(url1, url2);

Arguments

Name	Description
url1, url2	Two urls which will be compared.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
url_list={};
url_list[1]="http://10.10.10.10:80/"
url_list[2]="http://10.10.10.10/"
url_list[3]="https://5.5.5.5:443/"
url_list[4]="https://5.5.5.5/"
url_list[5]="http://[2001::1]:80"
url_list[6]="http://[2001::1]"
url_list[7]="https://[2001:99:1]:443"
url_list[8]="https://[2001:99:1]"
for i = 1,8,2 do
if url_compare(url_list[i],url_list[i+1]) then
debug("URL_List %d %d Match !\n",i,i+1);
else
debug("URL_List %d %d NOT Match !\n",i,i+1);
end
end
}
```

Supported Version

FortiADC version 5.2.x and later.

rand()

Generates a random number. Returns an integer number. After FortiADC reboots, the random number will be different.

Syntax

rand();

Arguments

N/A

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
a = rand()
debug(“a = %d\n”, a)
}
```

Supported Version

FortiADC version 5.2.x and later.

srand(str)

Sets the random seed.

Syntax

srand(str);

Arguments

Name	Description
str	A string which specifies the seed.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
srand(1111)
a = rand()
debug("a = %d\n", a)
}
```

Supported Version

FortiADC version 5.2.x and later.

rand_hex(int)

Generates a random number in HEX. Returns a string, length is the <int>.

Syntax

rand_hex(int);

Arguments

Name	Description
Int	An integer which specifies the length of the returned string.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
b = rand_hex(15)
debug("-----rand_hex b = %s-----\n", b)
}
Result:
-----rand_hex b = 43474FB47A8A8C4-----
```

Supported Version

FortiADC version 5.2.x and later.

rand_alphanum(int)

Generates a random alphabet + number sequence. Returns a string, length is the <int>.

Syntax

rand_alphanum(int);

Arguments

Name	Description
Int	An integer which specifies the length of the returned string.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
c = rand_alphanum(17)
debug("-----rand_alphanum c = %s-----\n", c)
}
Result:
-----rand_alphanum c = XTHQpb6ngabMqH7nx-----
```

Supported Version

FortiADC version 5.2.x and later.

rand_seq(int)

Generates a random number sequence. Returns a string, length is the <int>.

Syntax

rand_seq(int);

Arguments

Name	Description
Int	An integer which specifies the length of the returned string.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
d = rand_seq(18)
debug("-----rand_seq d = %s-----\n", d)
}
Result:

-----rand_seq = 329514876985314568-----
```

Supported Version

FortiADC version 5.2.x and later.

time()

Returns the current time as a number in seconds. This is the time since the Epoch was measured.

Syntax

time();

Arguments

N/A

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
t = time()
debug("-----time: t %s-----\n", t)
}
Result:
-----time: t 1561424783-----
```

Supported Version

FortiADC version 4.8.x and later.

ctime()

Returns the current time as a string, for example, “Tue Jun 25 14:11:01 2019”.

Syntax

ctime();

Arguments

N/A

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
ct = ctime()
debug(“-----ctime: ct %s-----\n”, ct)
}
Result:
-----ctime: ct Mon Jun 24 18:06:23 2019-----
```

Supported Version

FortiADC version 4.8.x and later.

gmtime()

Returns the GMT time as a string, for example, “Thu 27 Jun 2019 18:27:42 GMT”.

Syntax

gmtime();

Arguments

N/A

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
gt = gmtime()
debug(“-----gmtime: gt %s-----\n”, gt)
}
Result:
-----gmtime: gt Thu 27 Jun 2019 18:27:42 GMT -----
```

Supported Version

FortiADC version 5.3.x and later.

md5(str)

Returns the MD5 calculated for the specified string.

Syntax

md5(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “abc”
md5 = md5(str1)
str = “test string”
a=12
md = md5(“%s,123%d”,str,a)
}
```

Supported Version

FortiADC version 4.8.x and later.

md5_hex(str)

Returns the MD5 value in hex as a string.

Syntax

md5_hex(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “abc”
str2 = md5_hex(str1)
}
```

Supported Version

FortiADC version 4.8.x and later.

md5_str(str)

Calculates the MD5 of a string input and stores the results in an intermediate variable, in some cases you need a version to deal with it.

Syntax

md5_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
md5 = md5_str(input);     --input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

md5_hex_str(str)

Calculates the MD5 of a string input of a string input and outputs the results in HEX format, in some cases you need a version to deal with it.

Syntax

md5_hex_str(str);

Arguments

Name	Description
str	A string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
md = md5_hex_str(input);   --input  can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

sha1(str)

Returns the SHA-1 calculated for the specified string.

Syntax

sha1(str);

Arguments

Name	Description
str	A string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha1(input)
}
```

Supported Version

FortiADC version 4.8.x and later.

sha1_hex(str)

Returns the SHA-1 calculated for the string in hex.

Syntax

sha1_hex(str);

Arguments

Name	Description
str	A string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “123456789”
sha1 = sha1_hex(str1)
}
```

Supported Version

FortiADC version 4.8.x and later.

sha1_str(str)

Calculates the SHA-1 of a string input and stores the results in an intermediate variable, in some cases you need a version to deal with it.

Syntax

sha1_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha1_str(input);   --input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

sha1_hex_str(str)

Calculates the SHA-1 of a string input and output the results in HEX format, in some cases you need a version to deal with it.

Syntax

sha1_hex_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha1_hex_str(input);  -- input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

sha256(str)

Calculates the SHA-256 of a string input and stores the result in an intermediate variable.

Syntax

sha256(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “abc”
str2 = sha256(str1)
}
```

Supported Version

FortiADC version 4.8.x and later.

sha256_hex(str)

Calculates the SHA-256 of a string input and outputs the result in an intermediate variable. In some cases you need a version to deal with it.

Syntax

sha256_hex(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “abc”
sha256 = sha256_hex(str)
}
```

Supported Version

FortiADC version 4.8.x and later.

sha256_str(str)

Calculates the SHA-256 of a string input and stores the result in an intermediate variable. In some cases you need a version to deal with it.

Syntax

sha256_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha256_str(input);   --input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

sha256_hex_str(str)

Calculates the SHA-256 of a string input and stores the result in an intermediate variable. In some case you need a version to deal with it.

Syntax

sha256_hex_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha256_hex_str(input); --input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

sha384(str)

Calculates the SHA-384 of a string input and stores the result in an intermediate variable.

Syntax

sha384(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “abc”
str2 = sha384(str1)
}
```

Supported Version

FortiADC version 4.8.x and later.

sha384_hex(str)

Calculates the SHA-384 of a string input and outputs the result in an intermediate variable. In some cases you need a version to deal with it.

Syntax

sha384_hex(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “abc”
sha384 = sha384_hex(str)
}
```

Supported Version

FortiADC version 4.8.x and later.

sha384_str(str)

Calculates the SHA-384 of a string input and stores the result in an intermediate variable. In some cases you need a version to deal with it.

Syntax

sha384_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha384_str(input);   --input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

sha384_hex_str(str)

Calculates the SHA-384 of a string input and stores the result in an intermediate variable. In some case you need a version to deal with it.

Syntax

sha384_hex_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha384_hex_str(input); --input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

sha512(str)

Calculates the SHA-512 of a string input and stores the result in an intermediate variable.

Syntax

sha512(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “abc”
str2 = sha512(str1)
}
```

Supported Version

FortiADC version 4.8.x and later.

sha512_hex(str)

Calculates the SHA-512 of a string input and outputs the result in an intermediate variable. In some cases you need a version to deal with it.

Syntax

sha512_hex(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str1 = “abc”
sha512 = sha512_hex(str)
}
```

Supported Version

FortiADC version 4.8.x and later.

sha512_str(str)

Calculates the SHA-512 of a string input and stores the result in an intermediate variable. In some cases you need a version to deal with it.

Syntax

sha512_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha512_str(input);   --input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

sha512_hex_str(str)

Calculates the SHA-512 of a string input and stores the result in an intermediate variable. In some case you need a version to deal with it.

Syntax

sha512_hex_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = sha512_hex_str(input); --input can be a cert in DER format
}
```

Supported Version

FortiADC version 4.8.x and later.

b32_enc(str)

Encodes a string input in Base32 and outputs the result in string format.

Syntax

b32_enc(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str = “abc”
en = b32_enc(str)
}
```

Supported Version

FortiADC version 5.2.x and later.

b32_enc_str(str)

Encodes a string input in Base32 and outputs the result in string format. In some cases you need a version to deal with it.

Syntax

b32_enc_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = b32_enc_str(input);    --input can be a cert in DER format
}
```

Supported Version

FortiADC version 5.2.x and later.

b32_dec(str)

Decodes a Base32 encoded string input and outputs the result in string format.

Syntax

b32_dec(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str = “abc”
dec = b32_dec(str)
}
```

Supported Version

FortiADC version 5.2.x and later.

b32_dec_str(str)

Decodes a Base32 encoded string input and outputs the result in string format. In some cases you need a version to deal with it.

Syntax

b32_dec_str(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = b32_dec_str(input);    --input can be a cert in DER format
}
```

Supported Version

FortiADC version 5.2.x and later.

b64_enc(str)

Encodes a string input in Base64 and outputs the result in string format.

Syntax

b64_enc(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = b64_enc(input);
--Input can be general format:
str=”test string”
a=12
en=b64_enc(“%s, 123 %d”, str, a);
}
```

Supported Version

FortiADC version 5.2.x and later.

b64_dec(str)

Decodes a Base64 encoded string input and outputs the result in string format.

Syntax

b64_dec(str);

Arguments

Name	Description
str	The string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
result = b64_dec(input);
str=”test string”
a=12
de=b64_dec(“%s, 123 %d”, str, a);
}
```

Supported Version

FortiADC version 5.2.x and later.

get_pid()

Returns the PID value of the VS process.

Syntax

get_pid();

Arguments

N/A

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
pid = get_pid();
debug(“VS PID is : %d\n”, pid)
}
```

Supported Version

FortiADC version 5.2.x and later.

table_to_string(t)

Returns the table in a string.

Syntax

table_to_string(t);

Arguments

Name	Description
t	The table which specifies the information.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
t={};
t[1]=97;
t[2]=98;
t[3]=99;
t[4]=1;
str = table_to_string(t);
debug(“str is %s\n”, str)
}
Result:
str is abc
```

Supported Version

FortiADC version 4.8.x and later.

htonl(int)

Converts a 32 bit long integer from host byte order to network byte order.

Syntax

htonl(int);

Arguments

Name	Description
int	An integer which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str="0x12345678"
test=htonl(str)
debug("return : %x \n", test)
}
Result:
return: 78563412
```

Supported Version

FortiADC version 4.8.x and later.

ntohs(int)

Converts a 16 bit short integer from network byte order to host byte order.

Syntax

ntohs(int);

Arguments

Name	Description
int	An integer which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str="0x12345678"
test=ntohs(str)
debug("return : %x \n", test)
}
Result:
Return: 7856
```

Supported Version

FortiADC version 4.8.x and later.

htons(int)

Converts a 16 bit short integer from host byte order to network byte order.

Syntax

htons(int);

Arguments

Name	Description
int	An integer which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str="0x12345678"
test=htons(str)
debug("return : %x \n", test)
}
Result
Return: 7856
```

Supported Version

FortiADC version 4.8.x and later.

ntohl(int)

When receiving long integers in HTTP response from the network, this command converts a 32 bit long integer from network byte order to host byte order.

Syntax

ntohl(int);

Arguments

Name	Description
int	An integer which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str="0x12345678"
test=ntohl(str)
debug("return : %x \n", test)
log("record a log: %x \n", test)
}
Result:
return: 78563412
```

Supported Version

FortiADC version 4.8.x and later.

to_HEX(str)

Returns the HEX calculate of the string.

Syntax

to_HEX(str);

Arguments

Name	Description
str	A string which will be calculated.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
str = “\0\123\3”
hex = to_HEX(str)
debug(“this str in hex is: %s\n”, hex)
}
```

Supported Version

FortiADC version 4.8.x and later.

debug(str)

Prints the debug information when the virtual server is using HTTP scripting.

Syntax

debug(str);

Arguments

Name	Description
str	A string which will be printed.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
debug(“http request method is %s.\n”, HTTP:method_get())
}
```

Supported Version

FortiADC version 4.3.x and later.

log(str)

Prints the scripting running information in log format. When using this command, you should enable scripting log.

Syntax

log(str);

Arguments

Name	Description
str	A string which will be logged.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
log(“http request method is %s.\n”, HTTP:method_get())
}
```

Supported Version

FortiADC version 4.8.x and later.

file_open(path, str)

Opens a file, returns a file object.

Syntax

file_open(path, str);

Arguments

Name	Description
str	A string which specifies the method to open the file.
path	A string which specifies the file path.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
filepath = "/etc/resolv.conf";
fp = file_open(filepath,"r");
if not fp then
debug("file open failed\n");
end
repeat
line = file_gets(fp, 256);
if line then
debug("line %s", line);
end
until not line
file_close(fp);
}
```

Supported Version

FortiADC version 5.2.x and later.

file_gets(file, size)

Returns the file content.

Syntax

file_gets(file, size);

Arguments

Name	Description
file	A file object that get from file_open()

Events

Applicable in **all** events.

Supported Version

FortiADC version 5.2.x and later.

file_close(file)

Closes a file.

Syntax

file_close(file);

Arguments

Name	Description
file	A file object which will be closed.

Events

Applicable in **all** events.

Supported Version

FortiADC version 5.2.x and later.

GEO IP commands

- ip2country_name(ip) on page 101 – Returns the GEO information (country name) of an IP address.
- ip2countryProv_name(ip) on page 102 – Returns the GEO information (country name + possible province name) of an IP address.

ip2country_name(ip)

Returns the GEO information (country name) of an IP address.

Syntax

ip2country_name(ip);

Arguments

Name	Description
ip	A string which specifies the IP address.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
cip = IP:client_addr()
cnm = ip2country_name(cip)
debug(“cname %s\n”, cnm)
}
```

Supported Version

FortiADC version 5.2.x and later.

ip2countryProv_name(ip)

Returns the GEO information (country name + possible province name) of an IP address.

Syntax

ip2countryProv_name(ip);

Arguments

Name	Description
ip	A string which specifies the IP address.

Events

Applicable in **all** events.

Example

```
when HTTP_REQUEST {
cip = IP:client_addr()
cnm = ip2countryProv_name(cip)
debug(“cname %s\n”, cnm)
}
```

Supported Version

FortiADC version 5.2.x and later.

Management commands

Management (MGM) commands script event management functions:

- MGM:get_session_id() on page 104 – Returns the session ID.
- MGM:rand_id() on page 105 – Returns a 32-character string of HEX symbols.
- MGM:set_event(t) on page 106 – Allows the user to disable/enable the rest of the events from executing by disabling this event.
- MGM:set_auto(t) on page 107 – In the case of keep-alive, all events will be re-enabled automatically even if they are disabled in the previous TRANSACTION using the HTTP:set_event(t) command. To disable this automatic re-enabling behavior, you can call HTTP:set_auto(t).
- MGM:disable_event(code) on page 108 – Disables an event based on the code specified via the parameter.
- MGM:enable_event(code) on page 110 – Enables an event that was previously disabled based on the code specified via the parameter.
- MGM:disable_auto(code) on page 112 – Disables an event based on the code specified via the parameter.
- MGM:enable_auto(code) on page 114 – Enables an event that was previously disabled based on the code specified via the parameter.

MGM:get_session_id()

Returns the session ID.

Syntax

MGM:get_session_id();

Arguments

N/A

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, VS_LISTENER_BIND

Example

```
when HTTP_REQUEST {
sid = MGM:get_session_id()
debug(“session id: %s\n”, sid)
}
```

Supported Version

FortiADC version 5.0.x and later.

MGM:rand_id()

Returns a 32-character string of HEX symbols.

Syntax

MGM:rand_id();

Arguments

N/A

Events

Applicable in all events except VS_LISTENER_BIND and RULE_INIT.

Example

```
when HTTP_REQUEST {
rid = MGM:rand_id()
debug("rand id is %s\n", rid)
}
```

Supported Version

FortiADC version 5.0.x and later.

MGM:set_event(t)

Allows the user to disable/enable the rest of the events from executing by disabling this event.

Syntax

MGM:set_event(t);

Arguments

Name	Description
t	A table which specifies the event and operation.

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST.

Example

```
when HTTP_REQUEST {
t={};
t["event"]="req"; -- can be "req", "res", "data_req", "data_res", "ssl_client", "ssl_server",
"tcp_accept", "tcp_close", "ssl_renego_client", "ssl_renego_server", "server_connected", "server_
close", "server_before_connect", "vs_listener_bind", "auth_result", "cookie_bake"
t["operation"]="disable"; -- can be "enable", and "disable"
MGM:set_event(t);
debug("disable rest of the HTTP_REQUEST events\n");
}
```

Supported Version

FortiADC version 5.0.x and later.

MGM:set_auto(t)

In the case of keep-alive, all events will be re-enabled automatically even if they are disabled in the previous TRANSACTION using the HTTP:set_event(t) command. To disable this automatic re-enabling behavior, you can call HTTP:set_auto(t).

Syntax

MGM:set_auto(t);

Arguments

Name	Description
t	A table which specifies the event and operation to enable or disable.

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND.

Example

```
when HTTP_REQUEST {
t={};
t["event"]="req";
t["operation"]="disable";
MGM:set_auto(t);
debug("disable automatic re-enabling of the HTTP_REQUEST events\n");
}
```

Note: Event can be "req", "res", "data_req", "data_res", "ssl_server", "ssl_renego_server", "server_connected", "server_close", "server_before_connect." Operation can be "enable", and "disable."

Supported Version

FortiADC version 5.0.x and later.

MGM:disable_event(code)

Disables an event based on the code specified via the parameter.

Syntax

MGM:disable_event(code)

Arguments

Parameter	Description
code	A Lua integer in hex format to indicate the event. • 0x01 – EVENT_REQUEST_CODE • 0x02 – EVENT_RESPONSE_CODE • 0x04 –EVENT_DISBALE_DATA_REQUEST_CODE • 0x08 – EVENT_DISABLE_DATA_RESPONSE_CODE • 0x10 – EVENT_REQ_SSL_HANDSHAKE_CODE • 0x20 – EVENT_REP_SSL_HANDSHAKE_CODE • 0x40 – EVENT_TCP_ACCEPTED_CODE • 0x80 – EVENT_TCP_CLOSED_CODE • 0x100 – EVENT_REQ_SSL_RENEGOTIATE_CODE • 0x200 – EVENT_REP_SSL_RENEGOTIATE_CODE • 0x400 – EVENT_SERVER_CONNECTED_CODE • 0x800 – EVENT_SERVER_CLOSED_CODE • 0x1000 – EVENT_BEFORE_CONNECT_CODE • 0x2000 – EVENT_AUTH_RESULT_CODE • 0x4000 – EVENT_COOKIE_BAKE_CODE • 0x8000 – EVENT_PERSIST_CODE

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND

Example

In this example, code 0x1000 means BEFORE_CONNECT event, once that is disabled, the corresponding event function will not be called anymore.

```
when HTTP_REQUEST {
          MGM:disable_event(0x1000)
}
when SERVER_BEFORE_CONNECT {
--This will not be called.
    debug("------> Events: SERVER_BEFORE_CONNECT begin:[%s]\n", ctime())
    cip = IP:client_addr()
    debug("------> client IP %s\n", cip)
    debug("------> Events: SERVER_BEFORE_CONNECT end:[%s]\n", ctime())
}
```

Supported Version

FortiADC version 5.0.x and later.

MGM:enable_event(code)

Enables an event that was previously disabled based on the code specified via the parameter.

Syntax

MGM:enable_event(code)

Arguments

Parameter	Description
code	A Lua integer in hex format to indicate the event. • 0x01 – EVENT_REQUEST_CODE • 0x02 – EVENT_RESPONSE_CODE • 0x04 –EVENT_DISBALE_DATA_REQUEST_CODE • 0x08 – EVENT_DISABLE_DATA_RESPONSE_CODE • 0x10 – EVENT_REQ_SSL_HANDSHAKE_CODE • 0x20 – EVENT_REP_SSL_HANDSHAKE_CODE • 0x40 – EVENT_TCP_ACCEPTED_CODE • 0x80 – EVENT_TCP_CLOSED_CODE • 0x100 – EVENT_REQ_SSL_RENEGOTIATE_CODE • 0x200 – EVENT_REP_SSL_RENEGOTIATE_CODE • 0x400 – EVENT_SERVER_CONNECTED_CODE • 0x800 – EVENT_SERVER_CLOSED_CODE • 0x1000 – EVENT_BEFORE_CONNECT_CODE • 0x2000 – EVENT_AUTH_RESULT_CODE • 0x4000 – EVENT_COOKIE_BAKE_CODE • 0x8000 – EVENT_PERSIST_CODE

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND.

Example

In this example, the BEFORE_CONNECT event (code 0x1000) was previously disabled via MGM:disable_event(). Once code 0x1000 is enabled, the corresponding event function can be called again.

```
when HTTP_REQUEST {
          MGM:enable_event(0x1000)
}
when SERVER_BEFORE_CONNECT {
    debug("------> Events: SERVER_BEFORE_CONNECT begin:[%s]\n", ctime())
    cip = IP:client_addr()
    debug("------> client IP %s\n", cip)
    debug("------> Events: SERVER_BEFORE_CONNECT end:[%s]\n", ctime())
}
```

Supported Version

FortiADC version 5.0.x and later.

MGM:disable_auto(code)

This command is same as HTTP:disable_auto(), but it belongs to MGM class. And thus it can be called within more events.

Syntax

MGM:disable_auto(code);

Arguments

Name	Description
code	A LUA integer in hex format to indicate the event. The full list of events and codes are as below: Code Event Name 0x01 HTTP_REQUEST 0x02 HTTP_RESPONSE 0x04 HTTP_DATA_REQUEST 0x08 HTTP_DATA_RESPONSE 0x10 CLIENTSSL_HANDSHAKE 0x20 SERVERSSL_HANDSHAKE 0x40 TCP_ACCEPTED 0x80 TCP_CLOSED 0x100 CLIENTSSL_RENEGOTIATE 0x200 SERVERSSL_RENEGOTIATE 0x400 SERVER_CONNECTED 0x800 SERVER_CLOSED 0x1000 SERVER_BEFORE_CONNECT 0x2000 AUTH_RESULT 0x4000 COOKIE_BAKE 0x8000 PERSISTENCE 0x10000 BEFORE_AUTH 0x20000 POST_PERSIST 0x40000 WAF_REQUEST_BEFORE_SCAN 0x80000 WAF_RESPONSE_BEFORE_SCAN 0x100000 WAF_REQUEST_ATTACK_DETECTED 0x200000 WAF_RESPONSE_ATTACK_DETECTED 0x400000 VS_LISTENER_BIND

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND.

Example

```
when RULE_INIT {
count = 0
}
when HTTP_REQUEST {
count = count+1
if count>3 then
count=1
end
debug("==> begin REQUEST scripting: count=%d\n", count)
-- Disable RESPONSE event (code == 0x2)
code = 0x2
if (count == 1) then
debug("==> disable_event: count=%d\n", count)
MGM:disable_event(code)
--Also disable automatic enabling for the next request
MGM:disable_auto(code)
end
if (count == 2) then
-- Enable it for the third one.
MGM:enable_auto(code)
end
debug("==> end REQUEST scripting.\n\n")
}
when HTTP_RESPONSE {
debug("=====> begin RESPONSE scripting: count=%d\n", count)
debug("=====> end RESPONSE scripting.\n\n")
}
```

Note: Event can be "req", "res", "data_req", "data_res", "ssl_server", "ssl_renego_server", "server_connected", "server_close", "server_before_connect." Operation can be "enable", and "disable."

Supported Version

FortiADC version 5.0.x and later.

MGM:enable_auto(code)

This command is same as MGM:disable_auto(), but it does the opposite task.

By default, all the events are automatically enabled after disable_event() is called. So we only need to call this to undo earlier calling of disable_auto().

Syntax

MGM:enable_auto(code);

Arguments

Name	Description
code	A LUA integer in hex format to indicate the event. The full list of events and codes are as below: Code Event Name 0x01 HTTP_REQUEST 0x02 HTTP_RESPONSE 0x04 HTTP_DATA_REQUEST 0x08 HTTP_DATA_RESPONSE 0x10 CLIENTSSL_HANDSHAKE 0x20 SERVERSSL_HANDSHAKE 0x40 TCP_ACCEPTED 0x80 TCP_CLOSED 0x100 CLIENTSSL_RENEGOTIATE 0x200 SERVERSSL_RENEGOTIATE 0x400 SERVER_CONNECTED 0x800 SERVER_CLOSED 0x1000 SERVER_BEFORE_CONNECT 0x2000 AUTH_RESULT 0x4000 COOKIE_BAKE 0x8000 PERSISTENCE 0x10000 BEFORE_AUTH 0x20000 POST_PERSIST 0x40000 WAF_REQUEST_BEFORE_SCAN 0x80000 WAF_RESPONSE_BEFORE_SCAN 0x100000 WAF_REQUEST_ATTACK_DETECTED 0x200000 WAF_RESPONSE_ATTACK_DETECTED 0x400000 VS_LISTENER_BIND

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND

Example

```
when RULE_INIT {
count = 0
}
when HTTP_REQUEST {
count = count+1
if count>3 then
count=1
end
debug("==> begin REQUEST scripting: count=%d\n", count)
-- Disable RESPONSE event (code == 0x2)
code = 0x2
if (count == 1) then
debug("==> disable_event: count=%d\n", count)
MGM:disable_event(code)
--Also disable automatic enabling for the next request
MGM:disable_auto(code)
end
if (count == 2) then
-- Enable it for the third one.
MGM:enable_auto(code)
end
debug("==> end REQUEST scripting.\n\n")
}
when HTTP_RESPONSE {
debug("=====> begin RESPONSE scripting: count=%d\n", count)
debug("=====> end RESPONSE scripting.\n\n")
}
```

Note: Event can be "req", "res", "data_req", "data_res", "ssl_server", "ssl_renego_server", "server_connected", "server_close", "server_before_connect." Operation can be "enable", and "disable."

Supported Version

FortiADC version 5.0.x and later.

IP commands

IP commands contain functions to obtain IP layer related information such as obtaining the IP address and port of the server and client:

- IP:client_addr() on page 117 – Returns the client IP address of a connection; for the frontend, it will return the source address, while for the backend, it will return the destination address.
- IP:server_addr() on page 118 – Returns the IP address of the server in the backend.
- IP:local_addr() on page 120 – For the frontend, it returns the IP address of the virtual server that the client is connected to. For the backend, it returns the incoming interface IP address of the return packet.
- IP:remote_addr() on page 121 – Returns the IP address of the host on the far end of the connection.
- IP:client_port() on page 122 – Returns the client port number.
- IP:server_port() on page 123 – Returns the server port number. It is the real server port.
- IP:local_port() on page 125 – Returns the local port number. In the frontend, the local port is the virtual server port. In the backend, the local port is the port used to connect to the gateway.
- IP:remote_port() on page 126 – Returns the remote port number. In the frontend, the remote port is the client port. In the backend, remote port is the real server port.
- IP:client_ip_ver() on page 127 – Returns the current client IP version number of the connection, either 4 (for IPv4) or 6 (for IPv6).
- IP:server_ip_ver() on page 128 – Returns the current server IP version number of the connection, either 4 (for IPv4) or 6 (for IPv6).

IP:client_addr()

Returns the client IP address of a connection; for the frontend, it will return the source address, while for the backend, it will return the destination address.

Syntax

cip=IP:client_addr()

Arguments

N/A

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND, COOKIE_BAKE.

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:server_addr()

Returns the IP address of the server in the backend.

Syntax

sip=IP:server_addr()

Arguments

N/A

Events

Applicable in server-side events:

- `HTTP_DATA_RESPONSE`
- `HTTP_RESPONSE`
- `SERVER_CLOSED`
- `SERVER_CONNECTED`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:local_addr()

For the frontend, it returns the IP address of the virtual server that the client is connected to. For the backend, it returns the incoming interface IP address of the return packet.

Syntax

sip=IP:local_addr()

Arguments

N/A

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND, COOKIE_BAKE, SERVER_BEFORE_CONNECT.

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:remote_addr()

Returns the IP address of the host on the far end of the connection.

Syntax

sip=IP:remote_addr()

Arguments

N/A

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND, COOKIE_BAKE, SERVER_BEFORE_CONNECT.

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:client_port()

Returns the client port number.

Syntax

cp=IP:client_port()

Arguments

N/A

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND, COOKIE_BAKE.

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:server_port()

Returns the server port number. It is the real server port.

Syntax

sp=IP:server_port()

Arguments

N/A

Events

Applicable in server-side events:

- `HTTP_DATA_RESPONSE`
- `HTTP_RESPONSE`
- `SERVER_CLOSED`
- `SERVER_CONNECTED`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:local_port()

Returns the local port number. In the frontend, the local port is the virtual server port. In the backend, the local port is the port used to connect to the gateway.

Syntax

sp=IP:local_port()

Arguments

N/A

Events

`All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND, COOKIE_BAKE, SERVER_BEFORE_CONNECT.`

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:remote_port()

Returns the remote port number. In the frontend, the remote port is the client port. In the backend, remote port is the real server port.

Syntax

rp=IP:remote_port()

Arguments

N/A

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND, COOKIE_BAKE, SERVER_BEFORE_CONNECT.

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:client_ip_ver()

Returns the current client IP version number of the connection, either 4 (for IPv4) or 6 (for IPv6).

Syntax

cv=IP:client_ip_ver ()

Arguments

N/A

Events

All events except: CLIENTSSL_RENEGOTIATE, RULE_INIT, PERSISTENCE, POST_PERSIST, VS_LISTENER_BIND, COOKIE_BAKE.

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

IP:server_ip_ver()

Returns the current server IP version number of the connection, either 4 (for IPv4) or 6 (for IPv6).

Syntax

cv=IP:server_ip_ver ()

Arguments

N/A

Events

Applicable in server-side events:

- `HTTP_DATA_RESPONSE`
- `HTTP_RESPONSE`
- `SERVER_CLOSED`
- `SERVER_CONNECTED`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when SERVERSSL_HANDSHAKE {
cip=IP:client_addr()
lip=IP:local_addr()
sip=IP:server_addr()
rip=IP:remote_addr()
cp=IP:client_port()
lp=IP:local_port()
sp=IP:server_port()
rp=IP:remote_port()
sipv=IP:server_ip_ver();
cipv=IP:client_ip_ver();
debug("in server ssl with remote addr %s:%s client %s:%s, local %s:%s, server %s:%s, ip version
%s:%s\n", rip, rp, cip, cp, lip,lp, sip, sp, sipv, cipv)
}}
```

Supported Version

FortiADC version 5.0.x and later.

TCP commands

TCP commands contains functions to obtain and manipulate information related to the TCP layer, such as sockopt:

- TCP:reject() on page 131 – Allows the user to reject a TCP connection from a client.
- TCP:set_snat_ip(str) on page 132 – Allows the user to set the backend TCP connection's source address and port.
- TCP:clear_snat_ip() on page 134 – Allows the user to clear any IP that was set using the set_snat_ip() command.
- TCP:sockopt(t) on page 135 – Can set or get various socket/IP/TCP operations, such as buffer size, timeout, MSS, etc. For client-side events, this command applies to the client-side socket; for server-side events, it applies to server-side socket.
- TCP:after_timer_set(timer_cb_name, timeout, periodic) on page 141 – Allows the user to create and schedule a timer with a callback function and timeout value. This allows you to create multiple timers each with a unique callback function name.
- TCP:after_timer_cancel(timer_cb_name) on page 143 – Allows the user to cancel a scheduled timer. This function can only cancel a timer before it is triggered if it is not periodic.
- TCP:after_timer_get(timer_cb_name) on page 145 – Allows the user to get the information about the scheduled timers.
- TCP:close() on page 147 – Allows the user to close the TCP connection immediately.

TCP:reject()

Allows the user to reject a TCP connection from a client.

Syntax

TCP:reject();

Arguments

N/A

Events

Applicable in TCP_ACCEPTED.

Example

```
when TCP_ACCEPTED {
--check if the st is true or false;
if st then
TCP:reject();
end
}
```

Supported Version

FortiADC version 5.0.x and later.

TCP:set_snat_ip(str)

Allows the user to set the backend TCP connection's source address and port.

Syntax

TCP:set_snat_ip(str);

Note: To use the set_snat_ip() command, you must ensure the SOURCE ADDRESS flag is selected in the HTTP or HTTPS profile type.

Arguments

Name	Description
str	A string which specifies the ip address.

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `CLIENTSSL_RENEGOTIATE`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`
- `TCP_ACCEPTED`

Example

```
when TCP_ACCEPTED {
addr_group = “172.24.172.60/24”
client_ip = IP:client_addr()
matched = cmp_addr(client_ip, addr_group)
if matched then
if TCP:set_snat_ip(“10.106.3.124”) then
debug(“set SNAT ip to 10.106.3.124\n”)
end
end
}
```

Note: The VS must have the client address enabled in the profile, as shown in the example below.

```
config load-balance profile
   edit "http"
      set type http
      set client-address enable
```

```
    next
end
```

Supported Version

FortiADC version 5.2.x and later.

TCP:clear_snat_ip()

Allows the user to clear any IP that was set using the set_snat_ip() command.

Syntax

TCP:clear_snat_ip();

Arguments

N/A

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `CLIENTSSL_RENEGOTIATE`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`
- `TCP_ACCEPTED`

Example

```
when HTTP_REQUEST {
if TCP:clear_snat_ip() then
debug(“clear SNAT ip!\n”)
}
```

Supported Version

FortiADC version 5.0.x and later.

TCP:sockopt(t)

The TCP:sockopt() function allows you to set or retrieve various socket, IP, and TCP options, such as buffer size, timeout, and Maximum Segment Size (MSS).

Note: In FortiADC version 7.4.3, the TCP:sockopt() functionality has been extended with the added new "type" parameter that allows FortiADC to read the customized TCP option. However, the "type" parameter currently only supports the GET operations.

Syntax

TCP:sockopt(t)

Arguments

Name	Description
t	A Lua table specifying the operation and option.

The options and required parameters for the Lua table (t) depend on whether you are accessing Standard Options (using "message") or the Custom TCP Option (using "type").

1: Standard Options (Using "message")

Parameter Key	Operation	Required Value	Example	Notes
op	"get" or "set"	"get" or "set"	"set"	Specifies the action.
message	Both GET/SET	Option Name	"maxseg"	"snd_buf" "rcv_buf" "so_type" "so_priority" "ip_ttl" "fastopen" "maxseg" "so_mark" "tos" "ip_mtu" "tcp_nodelay" "tcp_cork" "tcp_quickack" "keepalive" "ip_reputation" "ipv6_v6only" "linger" "snd_timeo"

Parameter Key	Operation	Required Value	Example	Notes
				"rcv_timeo"
value	Only SET	New Option Value	1240	Required only for the "set" operation.

2: Custom TCP Option (Using "type")

Parameter Key	Operation	Required Value	Example	Notes
op	Only GET	"get"	"get"	Only the GET operation is supported.
type	Only GET	Option Number	28	Specifies the custom TCP option Type. Do not use "message".

Events

Applicable in the following events:

- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- HTTP_REQUEST
- HTTP_RESPONSE
- SERVER_BEFORE_CONNECT
- SERVER_CONNECTED
- TCP_ACCEPTED
- VS_LISTENER_BIND

Examples

The following examples and notes only apply to the "type" parameter that was introduced in V7.4.3:

```
when HTTP_REQUEST {
    debug("============begin scripting.\n")
    clientip = nil
    t = {}
    t["op"] = "get"
    -- Set the custom TCP option type
    t["type"] = 28
    ret = TCP:sockopt(t)
    if ret and (string.len(ret)>=4) then
        debug("------> TCP get sockopt(%d): (returned %d bytes) successfully.\n", t["type"],
string.len(ret));
        print_byte_array(ret)
        clientip = binStrToIpAddress(ret)
        debug("clientip = %s\n", clientip)
    else
        debug("------> TCP get sockopt(%d) failed.\n", t["type"])
    end

    if clientip then
        res = HTTP:header_insert("X-Forwarded-For", clientip)
        if res then
            debug("------> Header inserted successfully.\n")
        else
            debug("------> Header failed to insert.\n")
        end
    end
        debug("============end scripting.\n")
}

function binStrToIpAddress(binStr)
    return  tostring(string.byte(binStr,1)) .. "." .. tostring(string.byte(binStr,2)) ..
               "." .. tostring(string.byte(binStr,3)) .. "." .. tostring(string.byte(binStr,4))
end

function print_byte_array(s)
    for i=1, string.len(s) do
        debug("0x%x.", string.byte(s,i))
    end
    debug("\n")
end
```

Notes:

For TCP options including Kind 28 type packet, only the first 4 bytes will be read.

For example:

Sent:

```
Kind: 28
Length: …
192,168,1,100,192.168,1,200,192,168,1,2
```

FortiADC reads:

```
=========begin scripting.
clientip = 192.168.1.100
```

When the data sent is "abcd..." instead of regular numbers, decimals (ASCII code) 97,98,99,100 will be displayed.

For example:

Sent:

```
Kind: 28
Length: …
abcdef…
```

FortiADC reads:

```
------> TCP get sockopt(28): (returned 4 bytes) successfully.
0x61.0x62.0x63.0x64.
clientip = 97.98.99.100
```

If TCP options only contain two packets with Kind 28, only the first one will be read.

For example:

Sent:

```
Kind: 28
Length: 6
192,168,1,100
Kind: 28
Length: 6
192,168,1,100
```

FortiADC reads:

```
=========begin scripting.
clientip = 192.168.1.100
```

The following examples apply to the TCP:sockopt() as introduced in V5.0:

```
when RULE_INIT {
debug(" ======== RULE_INIT ========\n");
-- access to https://notes.shichao.io/unp/ch7/ for more details.
tcp_message = {};
tcp_message[1]="snd_buf"; --int
tcp_message[2]="rcv_buf"; --int
setIntMsg = {};
setIntMsg[1]="snd_buf"; --int
setIntMsg[2]="rcv_buf"; --int
setIntValue = {};
setIntValue[1] = 111222;
setIntValue[2] = 111222;
}
```

```
when VS_LISTENER_BIND {
--when a VS tries to bind.
debug(" ======== VS_LISTENER_BIND ========\n");
for k, v in pairs(tcp_message) do
t = {};
t["op"] = "get"
t["message"]=v
if TCP:sockopt(t) then
debug("%s value is %d\n",v, TCP:sockopt(t));
else
debug("get %s status     %s\n",v,TCP:sockopt(t));
end
end
debug("        ==== set ==== \n");
for k, v in pairs(setIntMsg) do
s = {};
s["op"] = "set"; --or "set"
s["message"] = v
s["value"] = setIntValue[k]; -- for integer value
result = TCP:sockopt(s);
debug("setting %s to %s return %s\n",v,setIntValue[k], result);
end
debug("        ==== End set ==== \n");
for k, v in pairs(tcp_message) do
t = {};
t["op"] = "get"
t["message"]=v
if TCP:sockopt(t) then
debug("%s value is %d\n",v, TCP:sockopt(t));
else
debug("get %s status     %s\n",v,TCP:sockopt(t));
end
end
}
```

```
when HTTP_RESPONSE {
debug(" ======== HTTP_RESPONSE ========\n");
t={}
t["size"] = 100;
HTTP:collect(t)
debug("        ==== set ==== \n");
for k, v in pairs(setIntMsg) do
s = {};
s["op"] = "set"; --or "set"
s["message"] = v
s["value"] = setIntValue[k]; -- for integer value
result = TCP:sockopt(s);
debug("setting %s to %s return %s\n",v,setIntValue[k], result);
end
debug("        ==== End set ==== \n");
for k, v in pairs(tcp_message) do
t = {};
t["op"] = "get"
t["message"]=v
if TCP:sockopt(t) then
debug("%s value is %d\n",v, TCP:sockopt(t));
else
debug("get %s status     %s\n",v,TCP:sockopt(t));
end
end
}
```

```
when HTTP_DATA_RESPONSE {
debug(" ======== HTTP_DATA_RESPONSE ========\n");
debug("        ==== set ==== \n");
for k, v in pairs(setIntMsg) do
s = {};
s["op"] = "set"; --or "set"
s["message"] = v
s["value"] = setIntValue[k]; -- for integer value
result = TCP:sockopt(s);
debug("setting %s to %s return %s\n",v,setIntValue[k], result);
end
debug("        ==== End set ==== \n");
for k, v in pairs(tcp_message) do
t = {};
t["op"] = "get"
t["message"]=v
if TCP:sockopt(t) then
debug("%s value is %d\n",v, TCP:sockopt(t));
else
debug("get %s status     %s\n",v,TCP:sockopt(t));
end
end
}
```

Supported Version

FortiADC version 5.0.x and later. Updated in version 7.4.3.

TCP:after_timer_set(timer_cb_name, timeout, periodic)

Allows the user to create and schedule a timer with a callback function and timeout value. This allows you to create multiple timers each with a unique callback function name. Periodic timers will be executed periodically until the associated session is closed or the after_timer is closed.

Returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

TCP:after_timer_set (timer_cb_name, timeout, periodic);

Arguments

Name	Description
timer_cb_name	A string of the callback function name. This is also the unique identification of a timer. This parameter is required.
timeout	An integer as the timeout value in milliseconds. This parameter is required.
periodic	A Boolean to indicate whether this timer is periodic. This parameter is required.

Events

Applicable in the following events:

- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `SERVER_BEFORE_CONNECT`
- `SERVER_CONNECTED`
- `TCP_ACCEPTED`

Example

```
when TCP_ACCEPTED {
    debug("[%s]------> TCP accepted begin:\n",ctime());
       local_count = 10
       cip = IP:client_addr();
       debug("[%s]------> client IP %s\n", ctime(),cip);
       cport = IP:client_port();
       debug("[%s]------> client Port %s\n", ctime(),cport);
       debug("[%s]------> local_count= %d\n", ctime(),local_count);
```

```
        debug("[%s]------> After function called here.\n",ctime());

        AFTER_TIMER_NAME = function ()
                debug("[%s]=======> After function call begin:\n",ctime())
                cport = IP:client_port();
                debug("[%s]=======> client Port %s\n",ctime(),cport);
                debug("[%s]=======> After function call end.\n",ctime())
        end

        TCP:after_timer_set("AFTER_TIMER_NAME", 5000, true);
}
```

When the client successfully creates a TCP connection, the script will be executed. The function will be executed every 5 seconds (5000 milliseconds) and print out the text.

FortiADC console debug output:

```
[Thu Jan 11 16:30:50 2024]------> TCP accepted begin:
[Thu Jan 11 16:30:50 2024]------> client IP 10.1.0.161
[Thu Jan 11 16:30:50 2024]------> client Port 37818
[Thu Jan 11 16:30:50 2024]------> local_count= 10
[Thu Jan 11 16:30:50 2024]------> After function called here.
[Thu Jan 11 16:30:55 2024]=======> After function call begin:
[Thu Jan 11 16:30:55 2024]=======> client Port 37818
[Thu Jan 11 16:30:55 2024]=======> After function call end.
[Thu Jan 11 16:31:00 2024]=======> After function call begin:
[Thu Jan 11 16:31:00 2024]=======> client Port 37818
[Thu Jan 11 16:31:00 2024]=======> After function call end.
```

Supported Version

FortiADC version 7.4.1 and later.

TCP:after_timer_cancel(timer_cb_name)

Allows the user to cancel a scheduled timer. This function can only cancel a timer before it is triggered if it is not periodic.

Returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

TCP:after_timer_cancel (timer_cb_name);

Arguments

Name	Description
timer_cb_name	A string to indicate the name of the timer to be canceled. This parameter is required.

Events

Applicable in the following events:

- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `SERVER_BEFORE_CONNECT`
- `SERVER_CONNECTED`
- `TCP_ACCEPTED`

Example

```
when TCP_ACCEPTED {
       AFTER_TIMER_NAME = function ()
       debug("[%s]====>After function call begin:\n",ctime());
       debug("[%s]====>After function call end.\n",ctime());
end
TCP:after_timer_set("AFTER_TIMER_NAME", 1000, true);
}
when HTTP_REQUEST {
    debug("[%s]------> Events: HTTP_REQUEST begin:\n", ctime());
    TCP:after_timer_cancel("AFTER_TIMER_NAME");
}
```

When the client successfully creates a TCP connection, the script will be executed. When the HTTP is requested, the AFTER_TIMER_NAME will be stopped.

FortiADC console debug output:

```
[Thu Oct  5 13:37:51 2023]====>After function call begin:
[Thu Oct  5 13:37:51 2023]====>After function call end.
[Thu Oct  5 13:37:52 2023]====>After function call begin:
[Thu Oct  5 13:37:52 2023]====>After function call end.
[Thu Oct  5 13:37:53 2023]------> Events: HTTP_REQUEST begin:
```

Supported Version

FortiADC version 7.4.1 and later.

TCP:after_timer_get(timer_cb_name)

Allows the user to get the information about the scheduled timers.

When successful, returns a string for one timer, a table of strings for multiple timers. For example, the returned string “'AFTER_TIMER_NAME':5000:periodic” shows the timer name, expiration in milliseconds and if it is periodic.

Returns nil for all failures.

Syntax

TCP:after_timer_get (timer_cb_name);

Arguments

Name	Description
timer_cb_name	A string to indicate the name of the timer. This parameter is optional. If this parameter is empty, then the function will get the information for all existing timers.

Events

Applicable in the following events:

- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `SERVER_BEFORE_CONNECT`
- `SERVER_CONNECTED`
- `TCP_ACCEPTED`

Example

```
when TCP_ACCEPTED {
       AFTER_TIMER_NAME = function ()
       debug("[%s]====>After function call begin:\n",ctime());
       debug("[%s]====>After function call end.\n",ctime());
end

TCP:after_timer_set("AFTER_TIMER_NAME", 1000, true);
ret = TCP:after_timer_get("AFTER_TIMER_NAME");
debug("after_timer_get success: %s\n", ret);
}
```

When the client successfully creates a TCP connection, the script will be executed. This function gets the **after_timer** information and prints it out on the first line.

FortiADC console debug output:

```
after_timer_get success: 'AFTER_TIMER_NAME':1000:periodic
[Thu Oct  5 12:36:34 2023]====>After function call begin:
[Thu Oct  5 12:36:34 2023]====>After function call end.
[Thu Oct  5 12:36:35 2023]====>After function call begin:
[Thu Oct  5 12:36:35 2023]====>After function call end.
```

Supported Version

FortiADC version 7.4.1 and later.

TCP:close()

Allows the user to close the TCP connection immediately. Once an associated session is closed, all the after_timers will be deleted.

Returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

TCP:close();

Arguments

N/A

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `SERVER_BEFORE_CONNECT`
- `SERVER_CONNECTED`
- `TCP_ACCEPTED`

Example

```
when TCP_ACCEPTED {
       AFTER_TIMER_NAME = function ()
       debug("[%s]====>After function call begin:\n",ctime());
       debug("[%s]====>After function call end.\n",ctime());
end

TCP:after_timer_set("AFTER_TIMER_NAME", 1000, true);
}
when HTTP_REQUEST {
    debug("[%s]------> Events: HTTP_REQUEST begin:\n", ctime());
    TCP:close();
}
```

When the client successfully creates a TCP connection, the script will be executed. When the HTTP is requested, the TCP connection will be closed.

Client side:

```
[root@Client1 ~]# curl http://10.1.0.15 -v
*   Trying 10.1.0.15:80...
* Connected to 10.1.0.15 (10.1.0.15) port 80 (#0)
> GET / HTTP/1.1
> Host: 10.1.0.15
> User-Agent: curl/8.0.1
> Accept: */*
>
* Empty reply from server
* Closing connection 0
curl: (52) Empty reply from server
```

Supported Version

FortiADC version 7.4.1 and later.

SSL commands

SSL commands contain functions for obtaining SSL related information, such as obtaining certificates and SNI:

- SSL:cipher() on page 150 – Returns the cipher in the handshake.
- SSL:version() on page 151 – Returns the SSL version in the handshake.
- SSL:alg_keysize() on page 153 – Returns the SSL encryption key size in the handshake.
- SSL:client_cert() on page 155 – Returns the status of client-certificate-verify, whether or not it is enabled.
- SSL:sni() on page 157 – Returns the SNI or false (if no SNI).
- SSL:npn() on page 159 – Returns the next protocol negotiation string or false (if no NPN).
- SSL:alpn() on page 160 – Allows you to get the SSL ALPN extension.
- SSL:session(t) on page 161 – Allows you to get SSL session ID, reuse the session, or remove it from the cache.
- SSL:cert(t) on page 162 – Allows you to get the certificate information between local or remote.
- SSL:peer_cert(str) on page 164 – Returns the peer certificate in different formats.
- SSL:disable(side_name) on page 165 – Disables SSL processing on either the client or server side when non-SSL traffic is expected or desired.
- SSL:cert_request() on page 168 – Requests the client certificate and verifies it. This command returns Boolean true if successful, otherwise, returns Boolean false.
- SSL:get_verify_depth() on page 169 – Gets the client certificate verify depth. This command returns the depth as an integer.
- SSL:renegotiate() on page 170 – Requests the client side SSL renegotiation. This command returns Boolean true if successful, otherwise, returns Boolean false. Note that if renegotiation fails, this request will fail; we recommend only using this command if renegotiation is necessary.
- SSL:set_verify_depth(depth) on page 171 – Sets the client certificate verify depth. This command returns Boolean true if successful, otherwise, returns Boolean false.

SSL:cipher()

Returns the cipher in the handshake.

Syntax

SSL:cipher();

Arguments

N/A

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
debug("client_handshake\n")
ci=SSL:cipher()
debug("Cipher: %s \n",ci)
}
Result: (if client send https request with cipher ECDHE-RSA-DES-CBC3-SHA)
Cipher: ECDHE-RSA-DES-CBC3-SHA
```

Supported Version

FortiADC version 5.0.x and later.

SSL:version()

Returns the SSL version in the handshake.

Syntax

SSL:version();

Arguments

N/A

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
debug("client handshake\n")
ver=SSL:version();
debug("SSL Version: %s \n",ver);
}
Result: (client send https request with various version)
client handshake
SSL Version: TLSv1
or
client handshake
SSL Version: TLSv1.1
or
client handshake
SSL Version: TLSv1.2
or
client handshake
SSL Version: SSLv3
```

Supported Version

FortiADC version 5.0.x and later.

SSL:alg_keysize()

Returns the SSL encryption key size in the handshake.

Syntax

SSL:alg_keysize();

Arguments

N/A

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
debug("client handshake\n")
ci=SSL:cipher()
key=SSL:alg_keysize()
debug("Cipher: %s\n",ci)
debug("Alg key size: %s \n",key)
}
Result: (client send https request with various ciphers)
client handshake
Cipher: ECDHE-RSA-RC4-SHA
Alg key size: 128
or
client handshake
Cipher: ECDHE-RSA-DES-CBC3-SHA
Alg key size: 168
or
client handshake
Cipher: EDH-RSA-DES-CBC-SHA
Alg key size: 56
or
client handshake
```

```
Cipher: ECDHE-RSA-AES256-GCM-SHA384
Alg key size: 256
```

Supported Version

FortiADC version 5.0.x and later.

SSL:client_cert()

Returns the status of client-certificate-verify, whether or not it is enabled.

Syntax

SSL:client_cert();

Arguments

N/A

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `CLIENTSSL_HANDSHAKE`
- `HTTP_REQUEST`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
debug("client handshake\n")
cc=SSL:client_cert()
debug("Client cert: %s \n",cc)
}
```

Result:

1. If not verify certificate is not set:
   ```
   Debug output:
   client handshake
   Client cert: false
   ```
2. If enabled verify in client-ssl-profile:
   ```
   config system certificate certificate_verify
     edit "verify"
       config  group_member
   ```

```
      edit 2
        set ca-certificate ca6
      next
    end
  next
end
config load-balance client-ssl-profile
  edit "csp"
    set client-certificate-verify verify
  next
end
debug output:
client handshake
Client cert: true
```

Supported Version

FortiADC version 5.0.x and later.

SSL:sni()

Returns the SNI or false (if no SNI).

Syntax

SSL:sni();

Arguments

N/A

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
debug("client handshake\n")
cc=SSL:sni();
debug("SNI: %s \n",cc);
}
```

Result:

Enable sni in client-ssl-profile

```
config load-balance client-ssl-profile
   edit "csp"
      set client-sni-required enable
   next
end
```

1. Client sends HTTPS request without SNI:

   ```
   [root@NxLinux certs]# openssl s_client -connect 5.1.1.100:443
   Debug output:
   Client handshake
   SNI: false
   ```

2. Client sends HTTPS request with SNI:

```
openssl s_client -connect 5.1.1.100:443 -servername 4096-rootca-rsa-server1
debug output :
client handshake
SNI: 4096-rootca-rsa-server1
```

Supported Version

FortiADC version 5.0.x and later.

SSL:npn()

Returns the next protocol negotiation string or false (if no NPN).

Syntax

SSL:npn();

Arguments

N/A

Events

Applicable in the following events:

- CLIENTSSL_HANDSHAKE
- SERVERSSL_HANDSHAKE
- SERVERSSL_RENEGOTIATE
- WAF_REQUEST_ATTACK_DETECTED
- WAF_REQUEST_BEFORE_SCAN

Example

```
when CLIENTSSL_HANDSHAKE {
npn = SSL:npn()
}
```

Supported Version

FortiADC version 5.0.x and later.

SSL:alpn()

Allows you to get the SSL ALPN extension.

Syntax

SSL:alpn();

Arguments

N/A

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
alpn = SSL:alpn()
}
```

Supported Version

FortiADC version 5.0.x and later.

SSL:session(t)

Allows you to get SSL session ID, reuse the session, or remove it from the cache.

Syntax

SSL:session(t);

Arguments

Name	Description
t	A table which specifies the operation to the session.

Events

Applicable in the following events:

- `CLIENTSSL_HANDSHAKE`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
t={}
t[“operation”] = “get_id”;  --can be “get_id” or “remove” or “reused”
sess_Id = SSL:session(t)
if sess_id then
id = to_HEX(sess_id)
debug(“client sess id %s\n”, id)
else
sess_id = “FALSE”
end
}
```

Supported Version

FortiADC version 5.0.x and later.

SSL:cert(t)

Allows you to get the certificate information between local or remote.

Syntax

SSL:cert(t);

Arguments

Name	Description
t	A table which specifies the certificate direction, and operation.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `CLIENTSSL_HANDSHAKE`
- `HTTP_REQUEST`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
debug("client handshake\n")
t={}
t["direction"]="remote";
t["operation"]="index";
t["idx"]=0;
t["type"]="info";
cert=SSL:cert(t)
if cert then
debug("client has cert\n")
end
for k, v in pairs(cert) do
if k=="serial_number" or k=="digest" then
debug("cert info name %s, value in HEX %s\n", k, to_HEX(v));
else
debug("cert info name %s, value %s\n", k, v);
end
end
}
```

Note:

- direction: local and remote. In CLIENTSSL_HANDSHAKE, local means FortiADC's cert, remote means client's cert.
- operation: index, count, issuer
- type: info, der, (pem)

This command returns a table that contains all the information in the certificate.
In the return, it contains: key_algorithm, hash, serial_number, not Before, not After, signature_algorithm, version, digest, issuer_name, subject_name, old_hash, pin-sha256, finger_print.

Supported Version

FortiADC version 5.0.x and later.

SSL:peer_cert(str)

Returns the peer certificate in different formats.

Syntax

SSL:peer_cert(str);

Arguments

Name	Description
str	A string which specifies the certificate format.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `CLIENTSSL_HANDSHAKE`
- `HTTP_REQUEST`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when CLIENTSSL_HANDSHAKE {
cder = SSL:peer_cert(“der”);   --for remote leaf certificate, the input parameter can be “info” or
“der” or “pem”
if cder then
hash = sha1_hex_str(cder)
debug(“whole cert sha1 hash is: %s\n”, hash)
end
}
```

Supported Version

FortiADC version 5.0.x and later.

SSL:disable(side_name)

Disables SSL processing on either the client or server side when non-SSL traffic is expected or desired.

Returns Boolean true if successful, otherwise, returns Boolean false.

This command only disables the SSL function on the current virtual server, and does not change any settings of the virtual server.

Before executing this command, ensure that HTTP connections are able to work in your virtual server environment.

Syntax

SSL:disable(side_name);

Arguments

Name	Description
side_name	A Lua string to indicate on which side the SSL will be disabled. You can input either of the following: • clientside • serverside This argument is optional. If it is not specified, FortiADC will determine which side to use based on the event where this API is called.

Events

Applicable in the following events.

Client side:

- TCP_ACCEPTED

Server side:

- HTTP_REQUEST
- BEFORE_AUTH
- AUTH_RESULT
- PERSISTENCE
- POST_PERSIST
- SERVER_BEFORE_CONNECT

Examples

```
--Client side must be TCP ACCEPTED
when TCP_ACCEPTED {
       debug("------> TCP accepted begin:\n");
       srcIP = IP:client_addr();
       srcPort = IP:client_port();
       debug("------> Client ip:port %s:%s\n", srcIP, srcPort);

       destIP = IP:local_addr();
       destPort = IP:local_port();
       debug("------> Local ip:port %s:%s\n", destIP, destPort);

       if tonumber(destPort) == 80 then
               ret = SSL:disable("clientside");
               if ret then
                       debug("------> SSL disable clientside successfully.\n");
               else
                       debug("------> SSL disable clientside failed.\n");
               end
       else
           debug("------> SSL disable clientside skipped.\n");
       end

       debug("------> TCP accepted end.\n");
}
```

```
--Server side can be called within many events
when HTTP_REQUEST {
    debug("------> HTTP Request begin:\n");
       srcIP = IP:client_addr();
       srcPort = IP:client_port();
       debug("------> Client ip:port %s:%s\n", srcIP, srcPort);

       destIP = IP:local_addr();
       destPort = IP:local_port();
       debug("------> Local ip:port %s:%s\n", destIP, destPort);

       if tonumber(destPort) == 80 then
               ret = SSL:disable("serverside");
               if ret then
                       debug("------> SSL disable serverside successfully.\n");
               else
                       debug("------> SSL disable serverside failed.\n");
               end
       else
           debug("------> SSL disable serverside skipped.\n");
       end

       debug("------> HTTP Request end.\n");
}
```

Supported Version

FortiADC version 7.4.3 and later.

SSL:cert_request()

Requests the client certificate and verifies it. This command returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

SSL:cert_request()

Arguments

N/A

Events

- BEFORE_AUTH
- HTTP_REQUEST

Example

```
when HTTP_REQUEST {
          ret = SSL:cert_request()
          if ret then
               debug("------> Client cert verified successfully.\n");
          else
               debug("------> Client cert failed to verify.\n");
          end
}
```

Supported Version

FortiADC version 5.0.x and later.

SSL:get_verify_depth()

Gets the client certificate verify depth. This command returns the depth as an integer.

Syntax

SSL:get_verify_depth()

Arguments

N/A

Events

- BEFORE_AUTH
- HTTP_REQUEST
- CLIENTSSL_HANDSHAKE
- SERVERSSL_HANDSHAKE
- SERVERSSL_RENEGOTIATE

Example

```
when HTTP_REQUEST {
          ret = SSL:get_verify_depth()
          debug("------> get_verify_depth=%d.\n", ret);
}
```

Supported Version

FortiADC version 5.0.x and later.

SSL:renegotiate()

Requests the client side SSL renegotiation. This command returns Boolean true if successful, otherwise, returns Boolean false. Note that if renegotiation fails, this request will fail; we recommend only using this command if renegotiation is necessary.

Syntax

SSL:renegotiate()

Arguments

N/A

Events

- BEFORE_AUTH
- HTTP_REQUEST

Example

```
when HTTP_REQUEST {
          ret = SSL:renegotiate()
          if ret then
               debug("------> requests client side SSL renegotiation successfully.\n");
          else
               debug("------> requests client side SSL renegotiation failed.\n");
          end
}
```

Supported Version

FortiADC version 5.0.x and later.

SSL:set_verify_depth(depth)

Sets the client certificate verify depth. This command returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

SSL:set_verify_depth(depth)

Arguments

Parameter	Description
depth	A Lua integer to indicate the depth.

Events

- `BEFORE_AUTH`
- `CLIENTSSL_HANDSHAKE`
- `HTTP_REQUEST`
- `SERVERSSL_HANDSHAKE`
- `SERVERSSL_RENEGOTIATE`

Example

```
when HTTP_REQUEST {
          depth = 2
          ret = SSL:set_verify_depth(depth)
          if ret then
               debug("------> set_verify_depth(%d) successfully.\n", depth);
          else
               debug("------> set_verify_depth(%d) failed.\n", depth);
          end
}
```

Supported Version

FortiADC version 5.0.x and later.

Authentication commands

Authentication (AUTH) commands contain functions related to authentication and login:

- AUTH:get_baked_cookie() on page 173 – Allows you to retrieve the baked cookie.
- AUTH:set_baked_cookie(cookie) on page 174 – Allows you to customize the cookie attribute of the baked cookie.
- AUTH:on_off() on page 175 – Returns whether authentication is required or not.
- AUTH:success() on page 177 – Returns whether authentication is successful or not.
- AUTH:form_based() on page 179 – Returns whether the authentication is HTTP form based or not.
- AUTH:user() on page 181 – Returns the user name in the authentication.
- AUTH:pass() on page 183 – Returns the password in the authentication.
- AUTH:usergroup(RealmName, UserGroupName) on page 185 – Returns the usergroup which the user belong to.
- AUTH:realm() on page 187 – Returns the realm in the authentication.
- AUTH:host() on page 189 – Returns the host in the authentication.
- AUTH:set_usergroup(RealmName, UserGroupName) on page 191 – Sets a new user group that is configured in the current authentication policy.
- AUTH:auth_flags() on page 192 – Gets authentication flags at the transaction level. This command returns the flags as an integer.
- AUTH:author_type() on page 193 – Gets the authentication behavior type. This command returns the type as an integer.
- AUTH:clt_meth() on page 194 – Gets the authentication client method. This command returns the method as an integer.
- AUTH:domain_prefix() on page 195 – Gets the authentication domain prefix. This command returns a string if successful, otherwise returns Boolean false.
- AUTH:flags() on page 196 – Gets authentication flags at the transaction level. This command returns the flags as an integer.
- AUTH:logoff() on page 197 – Gets the authentication log-off URI. This command returns a string if successful, otherwise returns Boolean false.
- AUTH:method() on page 198 – Gets the authentication method. This command returns the method as an integer.
- AUTH:relay_type() on page 199 – Gets the authentication relay type. This command returns the type as an integer.
- AUTH:result() on page 200 – Gets the authentication result. This command returns the result as an integer.
- AUTH:sso_domain() on page 201 – Gets the authentication SSO domain. This command returns a string if successful, otherwise returns Boolean false.
- AUTH:sso_group() on page 202 – Gets authentication the SSO group type. This command returns the result as an integer. It can be zero or one.
- AUTH:uri() on page 203 – Gets the authentication URI. This command returns a string if successful, otherwise returns Boolean false.

AUTH:get_baked_cookie()

Allows you to retrieve the baked cookie.

Syntax

AUTH:get_baked_cookie();

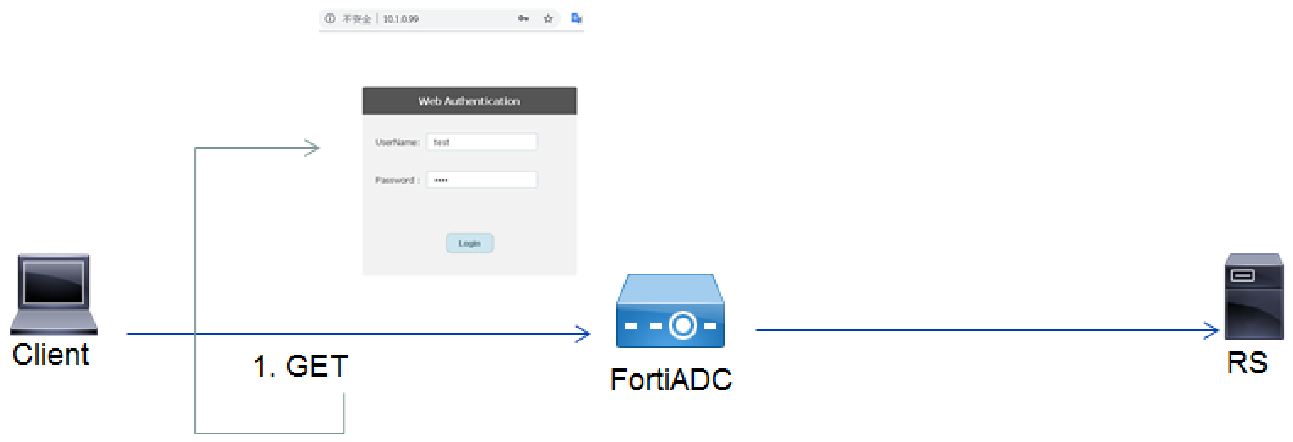
Arguments

N/A

Events

`COOKIE_BAKE`

Example



```
when COOKIE_BAKE {
cookie = AUTH:get_baked_cookie()
debug("Get cookie: %s\r\n", cookie)
}
Result:
Get cookie: Set-Cookie:
FortiADCauthSI=lfGnC2gsl7xsbAg4JFs94e4CJfFXaP3U5z6QHvo7n08cCoT5MdtQog2LmcizPo3aRiBHY/RThhocqG+Ddnv
sCLFJh3nBUoLeuYjGK9lY5L4=|W86hXGg; expires=Tue 23 Oct 2018 04:19:45 GMT; domain=10.1.0.99; path=/
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:set_baked_cookie(cookie)

Allows you to customize the cookie attribute of the baked cookie.

Syntax

AUTH:set_baked_cookie(cookie);

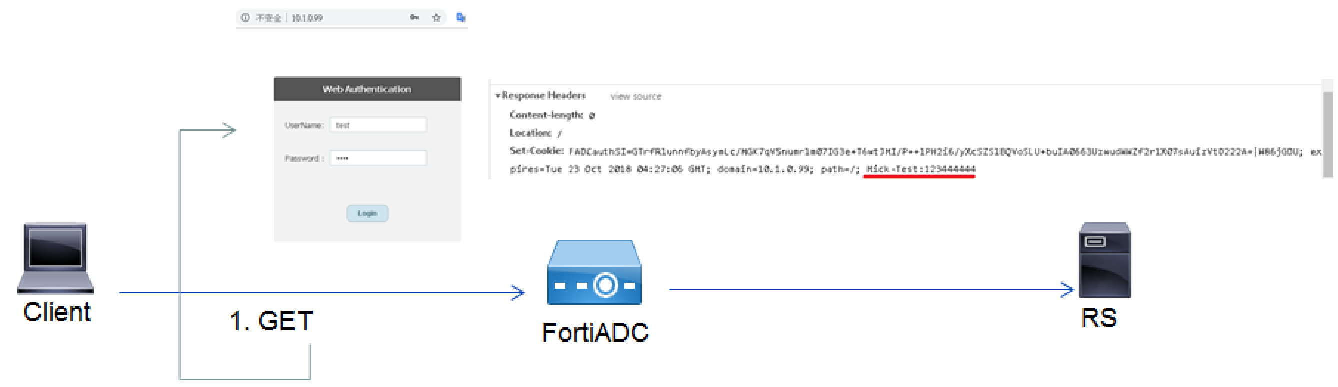
Arguments

Name	Description
cookie	A string which specifies the baked cookie.

Events

`COOKIE_BAKE`

Example



```
▾Response Headers    view source
  Content-length: 0
  Location: /
  Set-Cookie: FADCauthSI=GTrfRlunnfbyAsymLc/MGK7qVSnumr1m07IG3e+T6wtJHI/P++1PH2i6/yXcSZS1BQVoSLU+buIA0663UzwudWWZf2r1X07sAuizVtD222A=|W86jGDU; ex
  pires=Tue 23 Oct 2018 04:27:06 GMT; domain=10.1.0.99; path=/; Mick-Test:123444444
```

```
when COOKIE_BAKE {
cookie = AUTH:get_baked_cookie()
new_cookie = cookie..”; Mick-Test:123444444"
status = AUTH:set_baked_cookie(new_cookie)
debug(“Set baked cookie, status: %s\n”, status)
}
Result:
Set baked cookie, status: true
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:on_off()

Returns whether authentication is required or not.

Syntax

AUTH:on_off();

Arguments

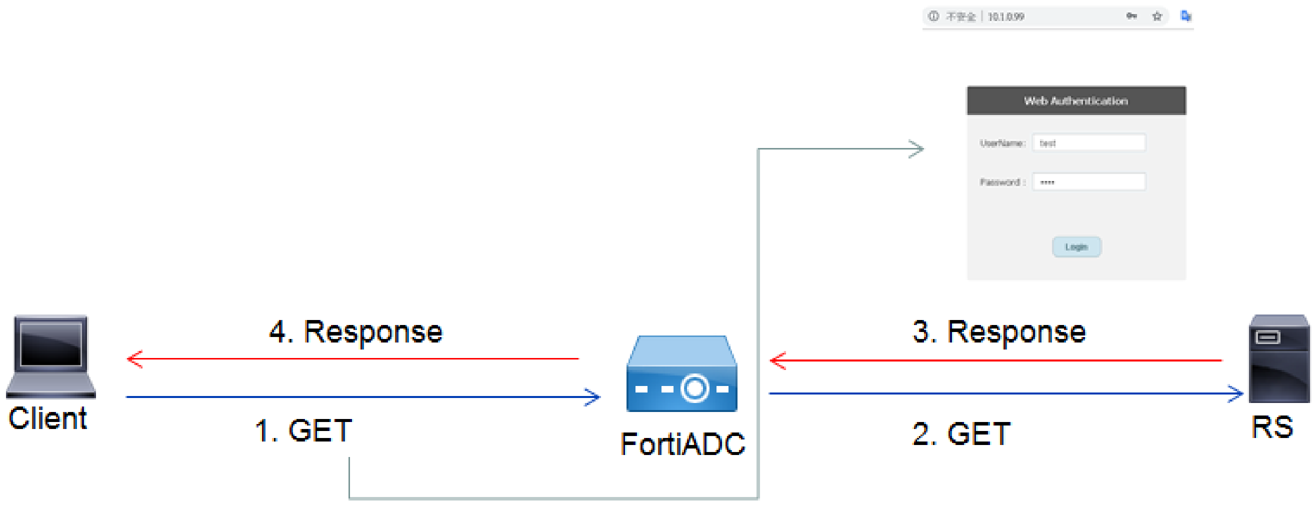
N/A

Events

Applicable in the following events:

- AUTH_RESULT
- HTTP_REQUEST
- HTTP_DATA_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_RESPONSE

Example



```
when AUTH_RESULT {
on_off = AUTH:on_off()
succ = AUTH:success()
fm = AUTH:form_based()
user = AUTH:user()
pass = AUTH:pass()
userg = AUTH:usergroup()
realm = AUTH:realm()
host = AUTH:host()
debug(“authentication form based %s, on_off %s, success %s, the user %s, pass %s, realm %s, the
usergroup %s, host %s\n”, fm, on_off, succ, user, pass, realm, userg, host)
}
Result:
authentication form based true, on_off true, success true, the user test, pass test, realm
Form333333, the userg test, host 10.1.0.99
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:success()

Returns whether authentication is successful or not.

Syntax

AUTH:success();

Arguments

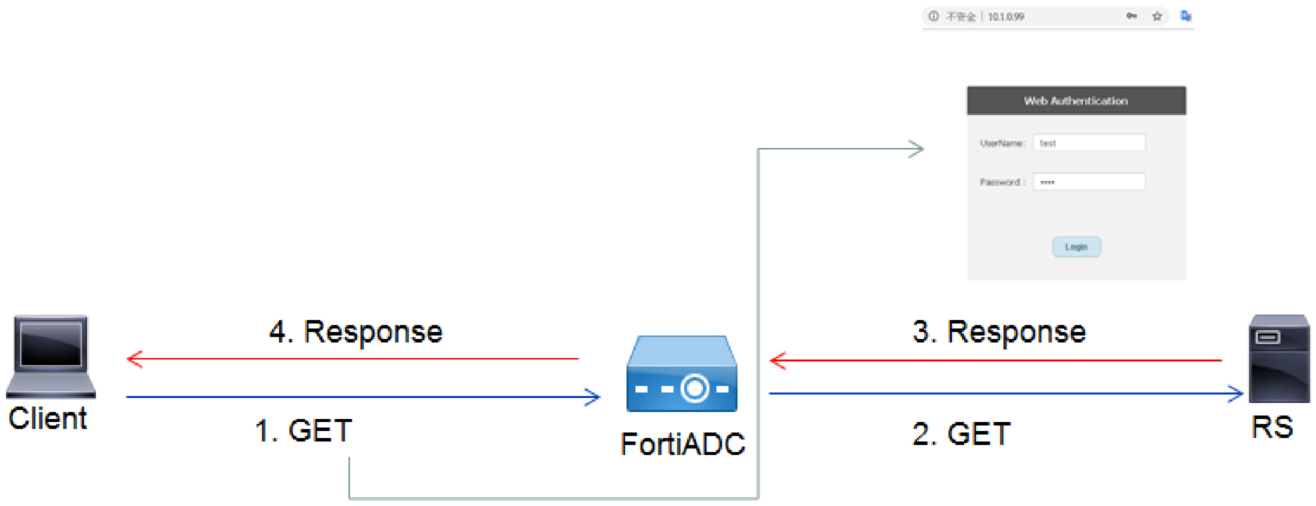
N/A

Events

Applicable in the following events:

- AUTH_RESULT
- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE

Example



```
when AUTH_RESULT {
on_off = AUTH:on_off()
succ = AUTH:success()
fm = AUTH:form_based()
user = AUTH:user()
pass = AUTH:pass()
userg = AUTH:usergroup()
realm = AUTH:realm()
host = AUTH:host()
debug("authentication form based %s, on_off %s, success %s, the user %s, pass %s, realm %s, the
usergroup %s, host %s\n", fm, on_off, succ, user, pass, realm, userg, host)
}
Result:
authentication form based true, on_off true, success true, the user test, pass test, realm
Form333333, the userg test, host 10.1.0.99
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:form_based()

Returns whether the authentication is HTTP form based or not.

Syntax

AUTH:form_based();

Arguments

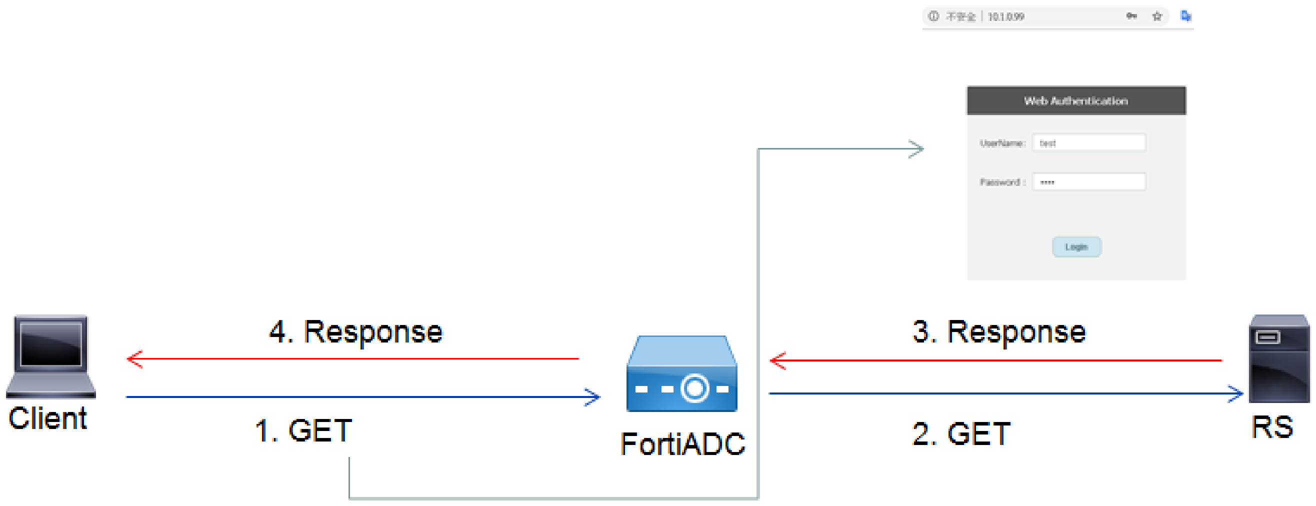
N/A

Events

Applicable in the following events:

- AUTH_RESULT
- HTTP_REQUEST
- HTTP_DATA_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_RESPONSE

Example



```
when AUTH_RESULT {
on_off = AUTH:on_off()
succ = AUTH:success()
fm = AUTH:form_based()
user = AUTH:user()
pass = AUTH:pass()
userg = AUTH:usergroup()
realm = AUTH:realm()
host = AUTH:host()
debug("authentication form based %s, on_off %s, success %s, the user %s, pass %s, realm %s, the
usergroup %s, host %s\n", fm, on_off, succ, user, pass, realm, userg, host)
}
Result:
authentication form based true, on_off true, success true, the user test, pass test, realm
Form333333, the userg test, host 10.1.0.99
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:user()

Returns the user name in the authentication.

Syntax

AUTH:user();

Arguments

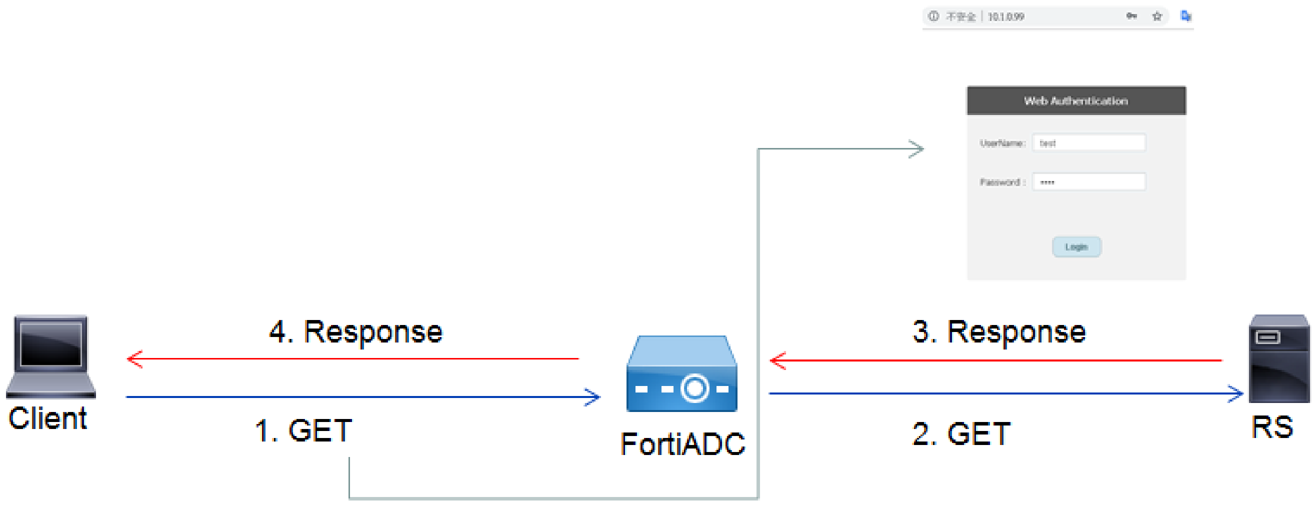
N/A

Events

Applicable in the following events:

- AUTH_RESULT
- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE

Example



```
when AUTH_RESULT {
on_off = AUTH:on_off()
succ = AUTH:success()
fm = AUTH:form_based()
user = AUTH:user()
pass = AUTH:pass()
userg = AUTH:usergroup()
realm = AUTH:realm()
host = AUTH:host()
debug("authentication form based %s, on_off %s, success %s, the user %s, pass %s, realm %s, the
usergroup %s, host %s\n", fm, on_off, succ, user, pass, realm, userg, host)
}
Result:
authentication form based true, on_off true, success true, the user test, pass test, realm
Form333333, the userg test, host 10.1.0.99
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:pass()

Returns the password in the authentication.

Syntax

AUTH:pass();

Arguments

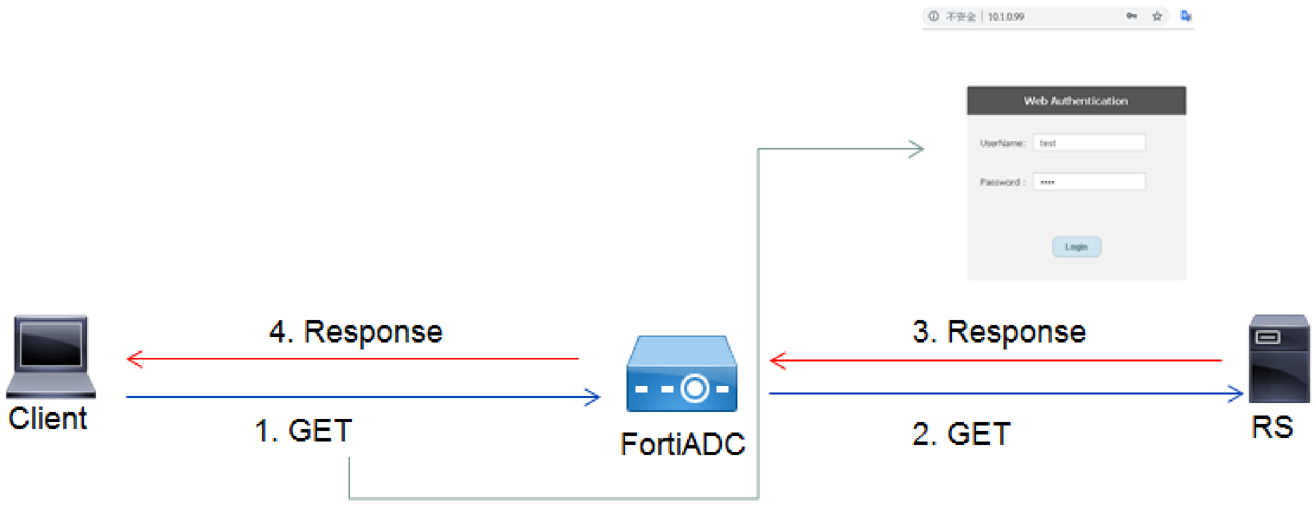
N/A

Events

Applicable in the following events:

- AUTH_RESULT
- HTTP_REQUEST
- HTTP_DATA_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_RESPONSE

Example



```
when AUTH_RESULT {
on_off = AUTH:on_off()
succ = AUTH:success()
fm = AUTH:form_based()
user = AUTH:user()
pass = AUTH:pass()
userg = AUTH:usergroup()
realm = AUTH:realm()
host = AUTH:host()
debug("authentication form based %s, on_off %s, success %s, the user %s, pass %s, realm %s, the
usergroup %s, host %s\n", fm, on_off, succ, user, pass, realm, userg, host)
}
Result:
authentication form based true, on_off true, success true, the user test, pass test, realm
Form333333, the userg test, host 10.1.0.99
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:usergroup(RealmName, UserGroupName)

Returns the user group which the user belong to.

Syntax

AUTH:usergroup(RealmName, UserGroupName);

Arguments

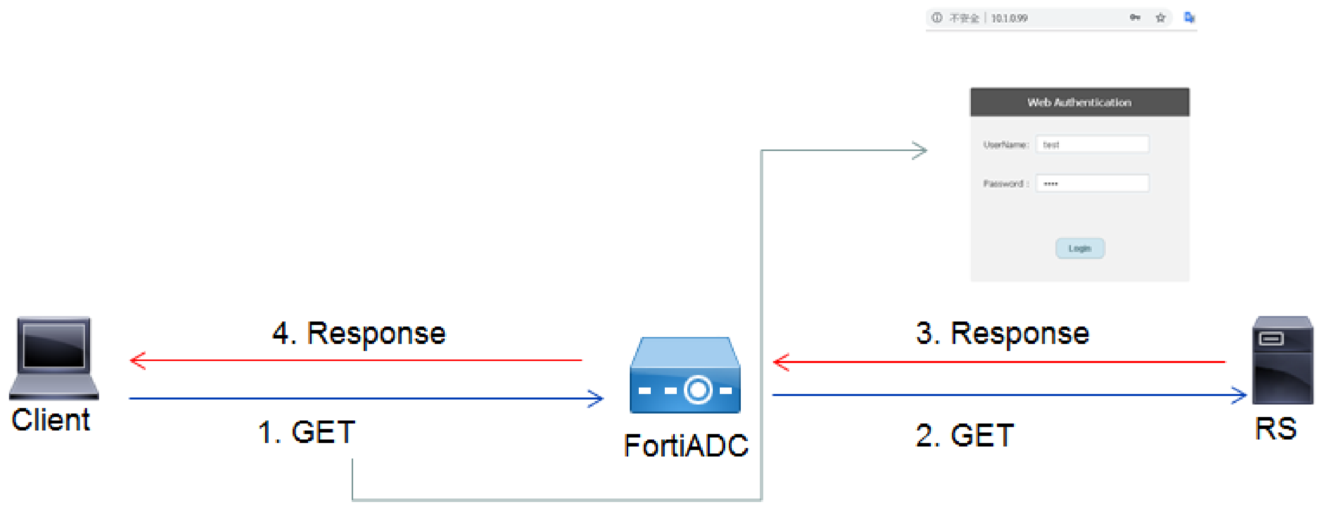
N/A

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `AUTH_RESULT`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`

Example



```
when AUTH_RESULT {
on_off = AUTH:on_off()
succ = AUTH:success()
fm = AUTH:form_based()
user = AUTH:user()
pass = AUTH:pass()
userg = AUTH:usergroup()
realm = AUTH:realm()
host = AUTH:host()
debug("authentication form based %s, on_off %s, success %s, the user %s, pass %s, realm %s, the
usergroup %s, host %s\n", fm, on_off, succ, user, pass, realm, userg, host)
}
Result:
authentication form based true, on_off true, success true, the user test, pass test, realm
Form333333, the userg test, host 10.1.0.99
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:realm()

Returns the realm in the authentication.

Syntax

AUTH:realm();

Arguments

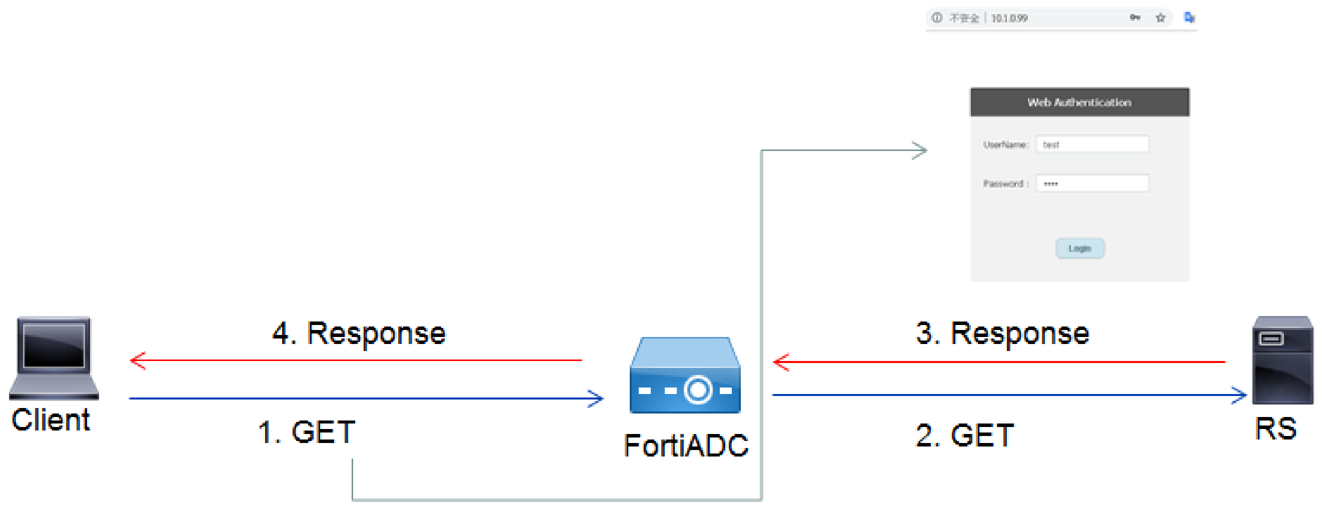
N/A

Events

Applicable in the following events:

- AUTH_RESULT
- BEFORE_AUTH
- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE

Example



```
when AUTH_RESULT {
on_off = AUTH:on_off()
succ = AUTH:success()
fm = AUTH:form_based()
user = AUTH:user()
pass = AUTH:pass()
userg = AUTH:usergroup()
realm = AUTH:realm()
host = AUTH:host()
debug("authentication form based %s, on_off %s, success %s, the user %s, pass %s, realm %s, the
usergroup %s, host %s\n", fm, on_off, succ, user, pass, realm, userg, host)
}
Result:
authentication form based true, on_off true, success true, the user test, pass test, realm
Form333333, the userg test, host 10.1.0.99
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:host()

Returns the host in the authentication.

Syntax

AUTH:host();

Arguments

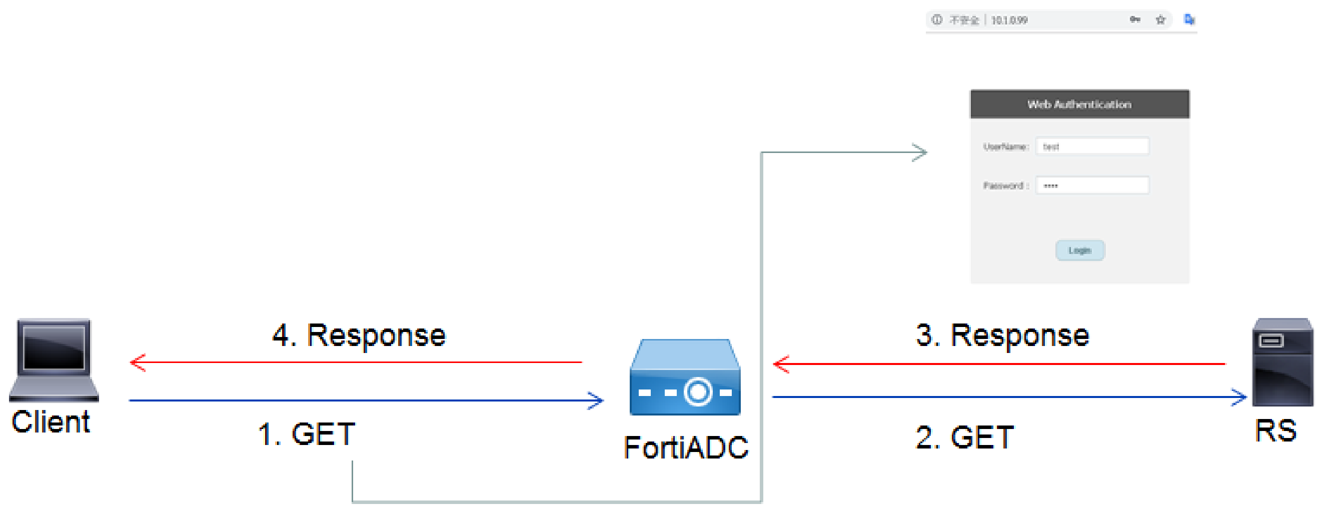
N/A

Events

Applicable in the following events:

- AUTH_RESULT
- BEFORE_AUTH
- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE

Example



```
when AUTH_RESULT {
on_off = AUTH:on_off()
succ = AUTH:success()
fm = AUTH:form_based()
user = AUTH:user()
pass = AUTH:pass()
userg = AUTH:usergroup()
realm = AUTH:realm()
host = AUTH:host()
debug(“authentication form based %s, on_off %s, success %s, the user %s, pass %s, realm %s, the
usergroup %s, host %s\n”, fm, on_off, succ, user, pass, realm, userg, host)
}
Result:
authentication form based true, on_off true, success true, the user test, pass test, realm
Form333333, the userg test, host 10.1.0.99
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:set_usergroup(RealmName, UserGroupName)

Sets a new user group that is configured in the current authentication policy. A new realm can also be set at the same time. It returns true if successful, otherwise, false. A realm name and a user group name are needed as input parameters.

The user group specified by the function must be in the authentication policy referenced by the VS. The result specified by the new user group will override the authentication result of the original authentication policy.

Syntax

AUTH:set_usergroup(RealmName, UserGroupName);

Arguments

Name	Description
RealmName	The name of the new realm to be set. (Lua string with maximum length of 63).
UserGroupName	The name of the user group to be set. (Lua string with maximum length of 63, must also comply with original definition of user group).

Events

`BEFORE_AUTH`

Example

```
when BEFORE_AUTH {
    r = AUTH:set_usergroup(“Realm02”, "UserGroup02");
   debug("set_usergroup successfully? %s\n", tostring(r));
}
```

Supported Version

FortiADC version 7.2.x and later.

AUTH:auth_flags()

Gets authentication flags at the transaction level. This command returns the flags as an integer.

Syntax

AUTH:auth_flags()

Arguments

N/A

Events

- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `AUTH_RESULT`

Example

```
when AUTH_RESULT {
    flags = AUTH:auth_flags()
    debug("===>>Authentication transaction flags=%d\n", flags)
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:author_type()

Gets the authentication behavior type. This command returns the type as an integer.

There are two types:

- 0 – AUTHOR_401
- 1 – AUTHOR_ALWAYS

Both types are used for HTTP basic Auth relay to describe the behavior when relaying. The authentication behavior type determines whether to send the HTTP authentication header after encountering a 401, or to send the HTTP authentication header regardless of encountering a 401.

Syntax

AUTH:author_type()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
    atype = AUTH:author_type()
debug("===>>authentication behavior type=%d\n", atype)
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:clt_meth()

Gets the authentication client method. This command returns the method as an integer.

There are four methods:

- 0 – CLT_HTTP
- 1 – CLT_FORM
- 2 – CLT_CERT
- 3 – CLT_NTLM

Syntax

AUTH:clt_meth()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
    meth = AUTH:clt_meth()
debug("===>>Authentication client method=%d\n", meth)
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:domain_prefix()

Gets the authentication domain prefix. This command returns a string if successful, otherwise returns Boolean false.

Syntax

AUTH:domain_prefix()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
    ret = AUTH:domain_prefix()
    if ret then
        debug("------> domain_prefix=%s.\n", ret)
    else
        debug("------> domain_prefix failed.\n")
end
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:flags()

Gets authentication flags at the session level. This command returns the flags as an integer.

Syntax

AUTH:flags()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
    flags = AUTH:flags()
    debug("===>>Authentication session flags=%d\n", flags)
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:logoff()

Gets the authentication log-off URI. This command returns a string if successful, otherwise returns Boolean false.

Syntax

AUTH:logoff()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
    ret = AUTH:logoff()
    if ret then
        debug("------> logoff=%s.\n", ret)
    else
        debug("------> logoff failed.\n")
end
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:method()

Gets the authentication method. This command returns the method as an integer.

The following methods are defined:

- -1 – HTTP_AUTH_WRONG
- 0 – HTTP_AUTH_UNKNOWN
- 1 – HTTP_AUTH_BASIC
- 2 – HTTP_AUTH_DIGEST
- 3 – HTTP_AUTH_FORMBASEING
- 4 – HTTP_AUTH_FORMBASED
- 5 – HTTP_AUTH_NTLM

Syntax

AUTH:method()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
       meth = AUTH:method()
debug("===>>Authentication method=%d\n", meth)
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:relay_type()

Gets the authentication relay type. This command returns the type as an integer.

The following relay types are defined:

- 0 – AUTH_RELAY_TYPE_KERBEROS
- 1 – AUTH_RELAY_TYPE_HTTP_BASIC
- 2 – AUTH_RELAY_TYPE_UNKNOWN

Syntax

AUTH:relay_type()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
       rt = AUTH:relay_type()
debug("===>>Authentication relay_type=%d\n", rt)
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:result()

Gets the authentication result. This command returns the result as an integer.

The following authentication results are defined:

- 2 – AUTH_CHALLENGE
- 1 – AUTH_SUCCESS
- 0 – AUTH_REJECT

Syntax

AUTH:result()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
       rt = AUTH:result ()
debug("===>>Authentication result=%d\n", rt)
}
```

Supported Version

FortiADC version 5.0.x and later.

AUTH:sso_domain()

Gets the authentication SSO domain. This command returns a string if successful, otherwise returns Boolean false.

Syntax

AUTH:sso_domain()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
    ret = AUTH:sso_domain()
    if ret then
        debug("------> sso_domain=%s.\n", ret)
    else
        debug("------> sso_domain failed.\n")
end
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:sso_group()

Gets authentication the SSO group type. This command returns the result as an integer. It can be zero or one.

Syntax

AUTH:sso_group()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
       ret = AUTH: sso_group()
debug("===>>Authentication sso_group=%d\n", ret)
}
```

Supported Version

FortiADC version 5.2.x and later.

AUTH:uri()

Gets the authentication URI. This command returns a string if successful, otherwise returns Boolean false.

Syntax

AUTH:uri()

Arguments

N/A

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- AUTH_RESULT

Example

```
when AUTH_RESULT {
    ret = AUTH: uri()
    if ret then
        debug("------> uri=%s.\n", ret)
    else
        debug("------> uri failed.\n")
end
}
```

Supported Version

FortiADC version 5.2.x and later.

HTTP commands

The HTTP scripting class includes basic functions for manipulating HTTP elements and other important functions:

- HTTP:header_get_names() on page 206 – Returns a list of all the headers present in the request or response.
- HTTP:header_get_values(header_name) on page 207 – Returns a list of value(s) of the HTTP header named <header_name>, with a count for each value.
- HTTP:header_get_value(header_name) on page 208 – Returns the value of the HTTP header named <header_name>.
- HTTP:header_remove(header_name) on page 209 – Removes all headers named with the name <header_name>.
- HTTP:header_remove2(header_name, countid) on page 210 – Returns a count ID for each item.
- HTTP:header_insert(header_name, value) on page 212 – Inserts the header <header_name> with value <value> into the end of the HTTP request or response.
- HTTP:header_replace(header_name, value) on page 213 – Replaces the occurrence value of header <header_name> with value <value>.
- HTTP:header_replace2(header_name, new-value, countid) on page 214 – Returns a count ID for each item for header_get_values().
- HTTP:header_exists(header_name) on page 216 – Returns true when the header <header_name> exists and false when it does not exist.
- HTTP:header_count(header_name) on page 217 – Returns the integer counter of the header <header_name>.
- HTTP:method_get() on page 218 – Returns the string of the HTTP request method.
- HTTP:method_set(str) on page 219 – Sets the HTTP request method to the string <value>.
- HTTP:path_get() on page 220 – Returns the string of the HTTP request path.
- HTTP:path_set(value) on page 221 – Sets HTTP request path to the string <value>.
- HTTP:uri_get() on page 222 – Returns the string of the HTTP request URI.
- HTTP:uri_set(value) on page 223 – Sets the HTTP request URI to the string <value>.
- HTTP:query_get() on page 224 – Returns the string of the HTTP request query.
- HTTP:query_set(value) on page 225 – Sets the HTTP request query to the string <value>.
- HTTP:redirect(str_url_withreplace [, parameter, ...]) on page 226 – Redirects an HTTP request or response to the specified URL.
- HTTP:redirect_with_cookie(url, cookie) on page 227 – Redirects an HTTP request or response to the specified URL with cookie.
- HTTP:redirect_t(t) on page 228 – Redirects an HTTP request or response to the URL specified in the table.
- HTTP:version_get() on page 230 – Returns the HTTP version of the request or response.
- HTTP:version_set(value) on page 231 – Sets the HTTP request or response version to the string <value>.
- HTTP:status_code_get() on page 232 – Returns the response status code output as string.
- HTTP:status_code_set(value) on page 233 – Sets the HTTP response status code.
- HTTP:code_get() on page 234 – Returns the response status code, output as integer.
- HTTP:code_set(integer) on page 235 – Sets the response status code.
- HTTP:reason_get() on page 236 – Returns the response reason.
- HTTP:reason_set(value) on page 237 – Sets the response reason.
- HTTP:client_addr() on page 238 – Returns the client IP address of a connection.

- HTTP:local_addr() on page 239 – For HTTP_REQUEST, returns the IP address of the virtual server the client is connected to; for HTTP_RESPONSE, returns the incoming interface IP address of the return packet.
- HTTP:server_addr() on page 240 – Returns the IP address of the server in HTTP_RESPONSE.
- HTTP:remote_addr() on page 241 – Returns the IP address of the host on the far end of the connection.
- HTTP:client_port() on page 242 – Returns real client port number in a string format.
- HTTP:local_port() on page 244 – Returns the local port number in a string format.
- HTTP:remote_port() on page 246 – Returns the remote port number in a string format.
- HTTP:server_port() on page 248 – Returns the real server port number in a string format.
- HTTP:client_ip_ver() on page 249 – Returns the client IP version number. This can be used to get IPv4 or IPv6 versions.
- HTTP:server_ip_ver() on page 250 – Returns the server IP version number. This can be used to get IPv4 or IPv6 versions.
- HTTP:close() on page 251 – Close an HTTP connection using code 503.
- HTTP:respond(t) on page 253 – Allows you to return a customized page and send out an HTTP response directly from FortiADC.
- HTTP:respond_errorfile(filename) on page 254 – Allows the HTTP to respond with a specified error file. This function returns Boolean true if successful otherwise, returns Boolean false.
- HTTP:get_session_id() on page 255 – FortiADC will assign each session a unique ID and allow the user to get this unique ID through this function.
- HTTP:rand_id() on page 257 – Returns a random string of 32 characters long in hex format.
- HTTP:set_event(t) on page 258 – Sets a request or response event to enable or disable.
- HTTP:set_auto() on page 260 – Sets an automatic request or response event.
- HTTP:get_unique_session_id() on page 263 – Returns a unique ID per session in history. Each ID is a string consisting of 32 hex digits, for example "b6e49ca3c937baaa47f2d111f593be8b".
- HTTP:get_unique_transaction_id() on page 262 – Returns a unique ID per transaction in history. Each ID is a string consisting of 32 hex digits, for example "b0b9fec0b4b28306a2bf4f63eb97520e".
- HTTP:disable_event(code) on page 264 – Disables an event based on the code specified via the parameter.
- HTTP:enable_event(code) on page 266 – Enables an event that was previously disabled based on the code specified via the parameter.
- HTTP:disable_auto(code) on page 268 – Enbles an event that was previously disabled based on the code specified via the parameter.
- HTTP:enable_auto(code) on page 271 – Enables an event that was previously disabled based on the code specified via the parameter.

HTTP:header_get_names()

Returns a list of all the headers present in the request or response.

Syntax

HTTP:header_get_names();

Arguments

N/A

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
--use header and value
headers = HTTP:header_get_names()
for k, v in pairs(headers) do
debug("The value of header %s is %s.\n", k, v)
end
--only use the header name
for name in pairs(headers) do
debug("The request/response includes header %s.\n",name)
end
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:header_get_values(header_name)

Returns a list of value(s) of the HTTP header named <header_name>, with a count for each value. Note that the command returns all the values in the headers as a list if there are multiple headers with the same name.

Syntax

HTTP:header_get_values(header_name);

Arguments

Name	Description
Header_name	A string which specifies the header name.

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
cookies=HTTP:header_get_values("Cookie")
for k, cnt in pairs(cookies) do
debug("initially include cookie %s cnt %d\n", k, v)
end
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:header_get_value(header_name)

Returns the value of the HTTP header named <header_name>.

Returns false if the HTTP header named <header_name> does not exist. The command operates on the value of the last head if there are multiple headers with the same name.

Syntax

HTTP:header_get_value(header_name);

Arguments

Name	Description
Header_name	A string which specifies the header name.

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
host = HTTP:header_get_value("Host");
debug("host is %s\n", host);
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:header_remove(header_name)

Removes all headers named with the name <header_name>.

Syntax

HTTP:header_remove(header_name);

Arguments

Name	Description
Header_name	A string which specifies the header name.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
HTTP:header_remove(“Cookie”);
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:header_remove2(header_name, countid)

This function removes a certain header of a given name by referencing its unique position using the countid. The countid is an integer that specifies the header's serial number, typically obtained from HTTP:header_get_values().

Syntax

HTTP:header_remove2(header_name, countid);

Arguments

Name	Description
Header_name	A string which specifies the header name.
Countid	A integer which specifies the header_name serial number

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
cookies=HTTP:header_get_values("Set-Cookie")
for k, v in pairs(cookies) do
debug("include cookie %s cnt %d\n", k, v)
end
if HTTP:header_remove2("Set-Cookie", 1) then
debug("remove 1st cookie\n")
end
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:header_insert(header_name, value)

Inserts the header <header_name> with value <value> into the end of the HTTP request or response.

Syntax

HTTP:header_insert(header_name, value);

Arguments

Name	Description
Header_name	A string which specifies the header name.
Value	A string which specifies the value of the header<header_name>.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
HTTP:header_insert(“Cookie”, “insert_cookie=server1”)
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:header_replace(header_name, value)

Replaces the occurrence value of header <header_name> with value <value>.

Syntax

HTTP:header_replace(header_name, value);

Arguments

Name	Description
Header_name	A string which specifies the header name.
Value	A string which specifies the value of the header<header_name>.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
HTTP:header_replace(“Host”, “www.fortinet.com”)
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:header_replace2(header_name, new-value, countid)

This function allows you to replace the value of a certain header of a given name by referencing its unique position using the countid. The countid is an integer that specifies the header's serial number, typically obtained from HTTP:header_get_values()

Syntax

HTTP:header_replace2(header_name, new-value, countid);

Arguments

Name	Description
Header_name	A string which specifies the header name.
new-value	The new string value for the specified header.
Countid	The 1-based index specifying which occurrence of the header to replace.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {debug("============begin scripting.\n")
    -- Replace the specified occurence of header name "HeaderTest"
    -- Both header name and value are plain strings, not regex
    count = 1
    ret = HTTP:header_replace2("HeaderTest", "New-Header-Value", count)
    if ret then
        debug("------> Header replaced successfully.\n");
    else
        debug("------> Header failed to replace.\n");
    end
```

```
        debug("============end scripting.\n")
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:header_exists(header_name)

Returns true when the header <header_name> exists and false when it does not exist.

Syntax

HTTP:header_exists(header_name);

Arguments

Name	Description
Header_name	A string which specifies the header name.

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
if HTTP:header_exists(“Cookie”) then
…
end
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:header_count(header_name)

Returns the integer counter of the header <header_name>.

Syntax

HTTP:header_count(header_name);

Arguments

Name	Description
Header_name	A string which specifies the header name.

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
count = HTTP:header_count(“Cookie”);
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:method_get()

Returns the string of the HTTP request method.

Syntax

HTTP:method_get();

Arguments

N/A

Events

Applicable in the following events:

- `PERSISTENCE`
- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
method = HTTP:method_get();
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:method_set(str)

Sets the HTTP request method to the string <value>.

Syntax

HTTP:method_set(str);

Arguments

Name	Description
str	A string which specifies the method.

Events

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
HTTP:method_set("POST")
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:path_get()

Returns the string of the HTTP request path.

Syntax

HTTP:path_get();

Arguments

N/A

Events

Applicable in the following events:

- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `PERSISTENCE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
path = HTTP:path_get();
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:path_set(value)

Sets the HTTP request path to the string <value>.

Syntax

HTTP:path_set(value);

Arguments

Name	Description
Value	A string which specifies the path.

Events

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
HTTP:path_set(“/other.html”);
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:uri_get()

Returns the string of the HTTP request URI.

Syntax

HTTP:uri_get();

Arguments

N/A

Events

Applicable in the following events:

- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `PERSISTENCE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
uri = HTTP:uri_get();
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:uri_set(value)

Sets the HTTP request URI to the string <value>.

Syntax

HTTP:uri_set(value);

Arguments

Name	Description
value	a string which specifices the uri.

Events

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
HTTP:uri_set("/index.html?para=xxxx");
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:query_get()

Returns the string of the HTTP request query.

Syntax

HTTP:query_get();

Arguments

N/A

Events

Applicable in the following events:

- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `PERSISTENCE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
query = HTTP:query_get();
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:query_set(value)

Sets the HTTP request query to the string <value>.

Syntax

HTTP:query_set(value);

Arguments

Name	Description
value	A string which specifies the URI.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
HTTP:query_set(“query1=value1”);
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:redirect(str_url_withreplace [, parameter, ...])

Redirects an HTTP request or response to the specified URL.

Syntax

HTTP:redirect(str_url_withreplace [, parameter, ...]);

Arguments

Name	Description
str_url_withreplace	A string defining the destination URL. This string can contain C-style format specifiers (e.g., %s, %d) that will be replaced by subsequent parameters.parameter: Optional. One or more variables or literal values used to replace the format specifiers found in str_url_withreplace. The number of parameters must match the number of specifiers.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Note: Cannot be used in HTTP_DATA_RESPONSE.

Example

```
when HTTP_REQUEST {
Host = HTTP:header_get_value("host")
Path = HTTP:path_get()
HTTP:redirect("https://%s%s", Host, Path);
}
```

Supported Version

FortiADC version 4.3.x and later.

HTTP:redirect_with_cookie(url, cookie)

Redirects an HTTP request or response to the specified URL with cookie.

Supports multiple redirect

Syntax

HTTP:redirect_with_cookie(url, cookie);

Arguments

Name	Description
url	A string which specifies the redirect url.
cookie	A string as cookie.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Note: Cannot be used in HTTP_DATA_RESPONSE.

Example

```
HTTP:redirect_with_cookie("www.example.com", "server=nginx")
HTTP:redirect_with_cookie("www.abc.com", "server=nginx")
```

The final redirection is to www.abc.com with the cookie "server=nginx".

Supported Version

FortiADC version 4.8.x and later.

HTTP:redirect_t(t)

Redirects an HTTP request or response to the URL specified in the table.

Supports multiple redirect, same as HTTP:redirect_with_cookie().

Syntax

HTTP:redirect_t(t);

Arguments

Name	Description
t	A table that defines the code, redirect url, and cookie.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Note: Cannot be used in HTTP_DATA_RESPONSE.

Example

```
when HTTP_RESPONSE {
a={}    --initialize a table
a["code"]=303;
a["url"]="www.example.com"
a["cookie"]="test:server"
HTTP:redirect_t(a)
debug("redirected\n")
}
```

Note:

- If the code is not set, then the default code in the HTTP redirect response will be 302.
- If the URL is missing in the input table, then a log will be generated.

Supported Version

FortiADC version 4.8.x and later.

HTTP:version_get()

Returns the HTTP version of the request or response.

Syntax

HTTP:version_get();

Arguments

N/A

Events

Applicable in the following events:

- BEFORE_AUTH
- HTTP_REQUEST
- HTTP_RESPONSE
- PERSISTENCE
- WAF_REQUEST_ATTACK_DETECTED
- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_ATTACK_DETECTED
- WAF_RESPONSE_BEFORE_SCAN

Example

```
when HTTP_REQUEST {
v = HTTP:version_get();
}
```

Supported Version

FortiADC version 5.8.x and later.

HTTP:version_set(value)

Sets the HTTP request or response version to the string <value>.

Syntax

HTTP:version_set(value);

Arguments

Name	Description
value	A string which specifies the version.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
HTTP:version_set(“HTTP2.0”);
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:status_code_get()

Returns the response status code output as string.

Syntax

HTTP:status_code_get();

Arguments

N/A

Events

- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
code = HTTP:status_code_get();
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:status_code_set(value)

Sets the HTTP response status code.

Syntax

HTTP:status_code_set(value);

Arguments

Name	Description
value	A string which specifies the status code.

Events

- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
HTTP:status_code_set(“304”);
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:code_get()

Returns the response status code, output as integer.

Syntax

HTTP:code_get();

Arguments

N/A

Events

- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
code = HTTP:code_get()
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:code_set(integer)

Sets the response status code.

Syntax

HTTP:code_set(integer);

Arguments

Name	Description
integer	Integer which specifies the status code.

Events

- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
HTTP:code_set(503);
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:reason_get()

Returns the response reason.

Syntax

HTTP:reason_get();

Arguments

N/A

Events

- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
reason = HTTP:reason_get();
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:reason_set(value)

Sets the response reason.

Syntax

HTTP:reason_set(value);

Arguments

Name	Description
value	A string which specifies the response reason.

Events

- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
HTTP:reason_set(“Not exist !”);
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:client_addr()

Returns the client IP address of a connection.

For HTTP_REQUEST packet, it's source address; for HTTP_RESPONSE packet, it's destination address.

Syntax

HTTP:client_addr();

Arguments

N/A

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `POST_PERSIST`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST priority 100 {
cip=HTTP:client_addr()
}
```

Supported Version

FortiADC version 4.6.x and later.

HTTP:local_addr()

For HTTP_REQUEST, returns the IP address of the virtual server the client is connected to; for HTTP_RESPONSE, returns the incoming interface IP address of the return packet.

Syntax

HTTP:local_addr();

Arguments

N/A

Events

Applicable in the following events:

- PERSISTENCE
- POST_PERSIST
- BEFORE_AUTH
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- HTTP_REQUEST
- HTTP_RESPONSE
- WAF_REQUEST_ATTACK_DETECTED
- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_ATTACK_DETECTED
- WAF_RESPONSE_BEFORE_SCAN

Example

```
when HTTP_REQUEST {
lip = HTTP:local_addr()
}
```

Supported Version

FortiADC version 4.6.x and later.

HTTP:server_addr()

Returns the IP address of the server in HTTP_RESPONSE.

Syntax

HTTP:server_addr();

Arguments

N/A

Events

- `HTTP_DATA_RESPONSE`
- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
sip = HTTP:server_addr()
}
```

Supported Version

FortiADC version 4.6.x and later.

HTTP:remote_addr()

Returns the IP address of the host on the far end of the connection

Syntax

HTTP:remote_addr();

Arguments

N/A

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `PERSISTENCE`
- `POST_PERSIST`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
rip = HTTP:remote_addr()
}
```

Supported Version

FortiADC version 4.6.x and later.

HTTP:client_port()

Returns the real client port number in a string format.

Syntax

HTTP:client_port();

Arguments

N/A

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `POST_PERSIST`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
string1=HTTP:client_port()
string2=HTTP:local_port()
string3=HTTP:remote_port()
debug("result_client_port: %s \n",string1)
debug("result_local_port: %s \n",string2)
debug("result_remote_port: %s \n",string3)
}
when HTTP_RESPONSE {
debug("SERVER_side: \n")
string4=HTTP:server_port()
debug("result_server_port: %s \n",string4)
string5=HTTP:client_port()
string6=HTTP:local_port()
string7=HTTP:remote_port()
debug("result_client_port: %s \n",string5)
debug("result_local_port: %s \n",string6)
debug("result_remote_port: %s \n",string7)
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:local_port()

Returns the local port number in a string format.

In HTTP_REQUEST, local_port is the virtual server port.

In HTTP_RESPONSE, local_port is the port of the gateway that was used to connect.

Syntax

HTTP:local_port();

Arguments

N/A

Events

Applicable in the following events:

- `PERSISTENCE`
- `POST_PERSIST`
- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
string1=HTTP:client_port()
string2=HTTP:local_port()
string3=HTTP:remote_port()
debug("result_client_port: %s \n",string1)
debug("result_local_port: %s \n",string2)
debug("result_remote_port: %s \n",string3)
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:remote_port()

Returns the remote port number in a string format.

In HTTP_REQUEST, remote_port is the client port.

In HTTP_RESPONSE, remote_port is the real server port.

Syntax

HTTP:remote_port();

Arguments

N/A

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `PERSISTENCE`
- `POST_PERSIST`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
string1=HTTP:client_port()
string2=HTTP:local_port()
string3=HTTP:remote_port()
debug("result_client_port: %s \n",string1)
debug("result_local_port: %s \n",string2)
debug("result_remote_port: %s \n",string3)
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:server_port()

Returns the real server port number in a string format.

Syntax

HTTP:server_port();

Arguments

N/A

Events

Applicable in the following events:

- `HTTP_DATA_RESPONSE`
- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
debug("SERVER_side: \n")
string4=HTTP:server_port()
debug("result_server_port: %s \n",string4)
string5=HTTP:client_port()
string6=HTTP:local_port()
string7=HTTP:remote_port()
debug("result_client_port: %s \n",string5)
debug("result_local_port: %s \n",string6)
debug("result_remote_port: %s \n",string7)
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:client_ip_ver()

Returns the client IP version number. This can be used to get IPv4 or IPv6 versions.

Syntax

HTTP:client_ip_ver();

Arguments

N/A

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `POST_PERSIST`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
string=HTTP:client_ip_ver()
debug("\nresult: %s \n",string)
}
when HTTP_RESPONSE {
string=HTTP:client_ip_ver()
debug("\nresult: %s \n",string)
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:server_ip_ver()

Returns the server IP version number. This can be used to get IPv4 or IPv6 versions.

Syntax

HTTP:server_ip_ver();

Arguments

N/A

Events

Applicable in the following events:

- `HTTP_DATA_RESPONSE`
- `HTTP_RESPONSE`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_RESPONSE {
string=HTTP:server_ip_ver()
debug("\nresult: %s \n",string)
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:close()

Close an HTTP connection using code 503.

Can support multiple close calls.

Syntax

HTTP:close();

Arguments

N/A

Events

```
Applicable in the following events:
```

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

```
Example
```

```
when HTTP_REQUEST {
    cnm = "strange"
    local patterns = {"strange"}
    st, count, t = class_match(cnm, "contains", patterns)
    debug("match status: %s, match count: %s\n", st, count)
    if st then
        -- Match found
        HTTP:close()
        debug("HTTP connection closed with 503 - Restricted\n")
    else
        debug("Normal traffic - allowed\n")
    end
}
```

Supported Version

FortiADC version 4.6.x and later.

HTTP:respond(t)

Allows you to return a customized page and send out an HTTP response directly from FortiADC.

Syntax

HTTP:respond(t);

Arguments

Name	Description
t	A table which will give the response code and content.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`
- `HTTP_RESPONSE supported since V7.2.4 and V7.4.1.`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
tt={}
tt["code"] = 200;
tt["content"] = "HTTP/1.1 200 OK\r\nConnection: close\r\nContent-Type: text/plain\r\n\r\nXXXXX
Test Page XXXXXXX";
status = HTTP:respond(tt);
debug("HTTP_respond() status: %s\n", status);
}
```

Supported Version

FortiADC version 5.2.x and later.

HTTP:respond_errorfile(filename)

Allows the HTTP to respond with a specified error file. This function returns Boolean true if successful otherwise, returns Boolean false.

Syntax

HTTP:respond_errorfile(filename);

Arguments

Parameter	Description
filename	A Lua string as the file name of the error file. The maximum length of a file name in Linux is 255 characters, including the file extension. Ensure the file name is valid, such as “403.html” or “path1/index.html”. If the file name is invalid, the API will still return as **true**, but the response will be **404 "Not Found"**. This parameter is mandatory.

Events

Applicable in the HTTP_REQUEST event.

Example

```
when HTTP_REQUEST {
    uri = HTTP:uri_get()
    filename = nil
    if uri=="/home/root" then
        filename = "403.html"
    …
     if filename ~= nil then
        HTTP:respond_errorfile(filename)
    end
}
```

Supported Version

FortiADC version 7.6.1 and later.

HTTP:get_session_id()

FortiADC will assign each session a unique ID and allow the user to get this unique ID through this function.

With this unique ID, the user can use Lua scripts to capture request headers, store them into a global variable indexed by this unique ID, and then index a global variable using this unique ID to extract its own request header information in the HTTP request event.

Syntax

HTTP:get_session_id();

Arguments

N/A

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
When RULE_INIT {
    env={}
}
when HTTP_REQUEST {
    id=HTTP:get_session_id()
    debug("session id %d\n", id);
    env[id]=nil
    req={}
    req["url"]=HTTP:uri_get()
    req["method"]=HTTP:method_get()
    env[id]=req
}
```

```
when HTTP_RESPONSE {
    id=1
    request=env[id]
    if req then
        debug("session id %d and url %s\n", id,request["url"]);
        debug("session id %d and method %s\n", id,request["method"]);
    end
}
Output:
session id 1
session id 1 and url /index.html
session id 1 and method GET
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:rand_id()

Returns a random string of 32 characters long in hex format.

Syntax

HTTP:rand_id();

Arguments

N/A

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
id = HTTP:rand_id();
debug(“random id: %s\n”, id)
}
```

Supported Version

FortiADC version 4.8.x and later.

HTTP:set_event(t)

Sets a request or response event to enable or disable.

Syntax

HTTP:set_event(t);

Arguments

Name	Description
t	A table that specifies when to enable/disable an event.

Events

Applicable in the following events:

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- BEFORE_AUTH
- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when HTTP_REQUEST {
t={};
t["event"] = "data_res";
t["operation"] = "disable";
HTTP:set_event(t);
}
```

Note:

The event can be "req", "res", "data_req", "data_res". And the operation can be "enable" and "disable". This command will generate a log if the event or operation is wrong.

Supported Version

FortiADC version 4.8.x and later.

HTTP:set_auto()

Sets an automatic request or response event.

In HTTP keep-alive mode, by default, FortiADC will automatically re-enable both HTTP_REQUEST and HTTP_RESPONSE event processes for the next transaction even if they have been disabled in the current transaction. Users can disable/enable this automatic behavior using this function.

Syntax

HTTP:set_auto(t);

Arguments

Name	Description
t	A table which specifies the event and operation.

Events

Applicable in the following events:

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- BEFORE_AUTH
- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when HTTP_REQUEST {
t={};
t["event"] = "data_req";
t["operation"] = "enable";
HTTP:set_auto(t);
}
```

Note:

The event can be "req", "res", "data_req", "data_res". And the operation can be "enable" and "disable". This command will generate a log if the event or operation is wrong.

Supported Version

FortiADC version 4.8.x and later.

HTTP:get_unique_transaction_id()

Returns a unique ID per transaction in history. Each ID is a string consisting of 32 hex digits, for example "b0b9fec0b4b28306a2bf4f63eb97520e".

Syntax

HTTP:get_unique_transaction_id();

Arguments

N/A

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
   tuid=HTTP:get_unique_transaction_id()
   debug("Unique transaction id %s\n", tuid);
}
```

Supported Version

FortiADC version 7.4.x and later.

HTTP:get_unique_session_id()

Returns a unique ID per session in history. Each ID is a string consisting of 32 hex digits, for example "b6e49ca3c937baaa47f2d111f593be8b".

Syntax

HTTP:get_unique_session_id();

Arguments

N/A

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_DATA_RESPONSE`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
   uid=HTTP:get_unique_session_id()
   debug("Unique session id %s\n", uid);
}
```

Supported Version

FortiADC version 7.4.x and later.

HTTP:disable_event(code)

Disables an event based on the code specified via the parameter.

Syntax

HTTP:disable_event(code)

Arguments

Parameter	Description
code	A Lua integer in hex format to indicate the event. • 0x01 – EVENT_REQUEST_CODE • 0x02 – EVENT_RESPONSE_CODE • 0x04 –EVENT_DISBALE_DATA_REQUEST_CODE • 0x08 – EVENT_DISABLE_DATA_RESPONSE_CODE • 0x10 – EVENT_REQ_SSL_HANDSHAKE_CODE • 0x20 – EVENT_REP_SSL_HANDSHAKE_CODE • 0x40 – EVENT_TCP_ACCEPTED_CODE • 0x80 – EVENT_TCP_CLOSED_CODE • 0x100 – EVENT_REQ_SSL_RENEGOTIATE_CODE • 0x200 – EVENT_REP_SSL_RENEGOTIATE_CODE • 0x400 – EVENT_SERVER_CONNECTED_CODE • 0x800 – EVENT_SERVER_CLOSED_CODE • 0x1000 – EVENT_BEFORE_CONNECT_CODE • 0x2000 – EVENT_AUTH_RESULT_CODE • 0x4000 – EVENT_COOKIE_BAKE_CODE • 0x8000 – EVENT_PERSIST_CODE

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- BEFORE_AUTH
- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

In this example, code 0x1000 means BEFORE_CONNECT event, once that is disabled, the corresponding event function will not be called anymore.

```
when HTTP_REQUEST {
          HTTP:disable_event(0x1000)
}
when SERVER_BEFORE_CONNECT {
--This will not be called.
    debug("------> Events: SERVER_BEFORE_CONNECT begin:[%s]\n", ctime())
    cip = IP:client_addr()
    debug("------> client IP %s\n", cip)
    debug("------> Events: SERVER_BEFORE_CONNECT end:[%s]\n", ctime())
}
```

Supported Version

FortiADC version 5.0.x and later.

HTTP:enable_event(code)

Enables an event that was previously disabled based on the code specified via the parameter.

Syntax

HTTP:enable_event(code)

Arguments

Parameter	Description
code	A Lua integer in hex format to indicate the event. • 0x01 – EVENT_REQUEST_CODE • 0x02 – EVENT_RESPONSE_CODE • 0x04 –EVENT_DISBALE_DATA_REQUEST_CODE • 0x08 – EVENT_DISABLE_DATA_RESPONSE_CODE • 0x10 – EVENT_REQ_SSL_HANDSHAKE_CODE • 0x20 – EVENT_REP_SSL_HANDSHAKE_CODE • 0x40 – EVENT_TCP_ACCEPTED_CODE • 0x80 – EVENT_TCP_CLOSED_CODE • 0x100 – EVENT_REQ_SSL_RENEGOTIATE_CODE • 0x200 – EVENT_REP_SSL_RENEGOTIATE_CODE • 0x400 – EVENT_SERVER_CONNECTED_CODE • 0x800 – EVENT_SERVER_CLOSED_CODE • 0x1000 – EVENT_BEFORE_CONNECT_CODE • 0x2000 – EVENT_AUTH_RESULT_CODE • 0x4000 – EVENT_COOKIE_BAKE_CODE • 0x8000 – EVENT_PERSIST_CODE

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- BEFORE_AUTH
- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

In this example, the BEFORE_CONNECT event (code 0x1000) was previously disabled via HTTP:disable_event(). Once code 0x1000 is enabled, the corresponding event function can be called again.

```
when HTTP_REQUEST {
          HTTP:enable_event(0x1000)
}
when SERVER_BEFORE_CONNECT {
    debug("------> Events: SERVER_BEFORE_CONNECT begin:[%s]\n", ctime())
    cip = IP:client_addr()
    debug("------> client IP %s\n", cip)
    debug("------> Events: SERVER_BEFORE_CONNECT end:[%s]\n", ctime())
}
```

Supported Version

FortiADC version 5.0.x and later.

HTTP:disable_auto(code)

This API disables the automatic enabling after disable_event() is called. It requires one parameter to indicate the event code. This command always returns Boolean true so there is no need to check that.

Syntax

HTTP:disable_auto(code)

Arguments

Parameter	Description
code	A LUA integer in hex format to indicate the event. 0x01 – HTTP_REQUEST 0x02 – HTTP_RESPONSE 0x04 – HTTP_DATA_REQUEST 0x08 – HTTP_DATA_RESPONSE 0x10 – CLIENTSSL_HANDSHAKE 0x20 – SERVERSSL_HANDSHAKE 0x40 – TCP_ACCEPTED 0x80 – TCP_CLOSED 0x100 – CLIENTSSL_RENEGOTIATE 0x200 – SERVERSSL_RENEGOTIATE 0x400 – SERVER_CONNECTED 0x800 – SERVER_CLOSED 0x1000 – SERVER_BEFORE_CONNECT 0x2000 – AUTH_RESULT 0x4000 – COOKIE_BAKE 0x8000 – PERSISTENCE 0x10000 – BEFORE_AUTH 0x20000 – POST_PERSIST 0x40000 – WAF_REQUEST_BEFORE_SCAN 0x80000 – WAF_RESPONSE_BEFORE_SCAN 0x100000 – WAF_REQUEST_ATTACK_DETECTED 0x200000 – WAF_RESPONSE_ATTACK_DETECTED 0x400000 – VS_LISTENER_BIND

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- BEFORE_AUTH
- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

In this example, code 0x2 means HTTP_RESPONSE event, once we disable it, the corresponding event function will not be called for this request. Once we call disable_auto() with the same code, it will continue disabled until enable_auto() is called.

```
when RULE_INIT {
    count = 0
}
when HTTP_REQUEST {
    count = count+1
    if count>3 then
        count=1
    end
    debug("==> begin REQUEST scripting: count=%d\n", count)
    -- Disable RESPONSE event (code == 0x2)
    code = 0x2
    if (count == 1) then
        debug("==> disable_event: count=%d\n", count)
        HTTP:disable_event(code)
        --Also disable automatic enabling for the next request
        HTTP:disable_auto(code)
    end
    if (count == 2) then
        -- Enable it for the third one.
        HTTP:enable_auto(code)
    end
    debug("==> end REQUEST scripting.\n\n")
}
when HTTP_RESPONSE {
    debug("=====> begin RESPONSE scripting: count=%d\n", count)
    debug("=====> end RESPONSE scripting.\n\n")
}
```

Supported Version

FortiADC version 5.0.x and later.

HTTP:enable_auto(code)

This command is same as HTTP:disable_auto(), but it does the opposite task. By default, all the events are automatically enabled after disable_event() is called. So we only need to call this to undo earlier calling of disable_auto (). See the example in disable_auto() section above.

Syntax

HTTP:enable_auto(code)

Arguments

Parameter	Description
code	A Lua integer in hex format to indicate the event. 0x01 – HTTP_REQUEST 0x02 – HTTP_RESPONSE 0x04 – HTTP_DATA_REQUEST 0x08 – HTTP_DATA_RESPONSE 0x10 – CLIENTSSL_HANDSHAKE 0x20 – SERVERSSL_HANDSHAKE 0x40 – TCP_ACCEPTED 0x80 – TCP_CLOSED 0x100 – CLIENTSSL_RENEGOTIATE 0x200 – SERVERSSL_RENEGOTIATE 0x400 – SERVER_CONNECTED 0x800 – SERVER_CLOSED 0x1000 – SERVER_BEFORE_CONNECT 0x2000 – AUTH_RESULT 0x4000 – COOKIE_BAKE 0x8000 – PERSISTENCE 0x10000 – BEFORE_AUTH 0x20000 – POST_PERSIST 0x40000 – WAF_REQUEST_BEFORE_SCAN 0x80000 – WAF_RESPONSE_BEFORE_SCAN 0x100000 – WAF_REQUEST_ATTACK_DETECTED 0x200000 – WAF_RESPONSE_ATTACK_DETECTED 0x400000 – VS_LISTENER_BINDT_REQUEST_CODE

Events

- HTTP_REQUEST
- HTTP_RESPONSE
- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE
- BEFORE_AUTH
- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

In this example, code 0x2 means HTTP_RESPONSE event, once we disable it, the corresponding event function will not be called for this request. Once we call disable_auto() with the same code, it will continue disabled until enable_auto() is called.

```
when RULE_INIT {
    count = 0
}
when HTTP_REQUEST {
    count = count+1
    if count>3 then
        count=1
    end
    debug("==> begin REQUEST scripting: count=%d\n", count)
    -- Disable RESPONSE event (code == 0x2)
    code = 0x2
    if (count == 1) then
        debug("==> disable_event: count=%d\n", count)
        HTTP:disable_event(code)
        --Also disable automatic enabling for the next request
        HTTP:disable_auto(code)
    end
    if (count == 2) then
        -- Enable it for the third one.
        HTTP:enable_auto(code)
    end
    debug("==> end REQUEST scripting.\n\n")
}
when HTTP_RESPONSE {
    debug("=====> begin RESPONSE scripting: count=%d\n", count)
    debug("=====> end RESPONSE scripting.\n\n")
}
```

Supported Version

FortiADC version 5.0.x and later.

HTTP DATA commands

If the HTTP data exceeds the 1.25M buffer size limit, FortiADC will only collect up to the 1.25M limit of data and forward the excess data directly.

- HTTP:collect(t) on page 275 – Collects body data from the HTTP request or response. You may specify a specific amount using the length argument.
- HTTP:payload(t) on page 277 – Manipulates or inspects the buffered HTTP request or response body. This function is a unified interface for performing various operations on the payload data, such as retrieving its size or content, searching, modifying, replacing, or removing parts of it.

HTTP:collect(t)

Collects body data from the HTTP request or response. You may specify a specific amount using the length argument.

Syntax

HTTP:collect(t)

Arguments

Name	Description
t	A table that specifies how much data to collect.

Operation

Instructs the system to buffer HTTP body data. This must be called before the data events (HTTP_DATA_REQUEST or HTTP_DATA_RESPONSE) can be processed.

Field	Type	Required	Description
size	Integer	Yes	The amount of HTTP body data to collect, in bytes.

Events

`Applicable in the following events:`

- `HTTP_REQUEST`
- `HTTP_RESPONSE`

Example

```
when HTTP_RESPONSE {
debug("Start\n")
t={}
t["size"]=10
HTTP:collect(t)
debug("Done\n")
}
when HTTP_DATA_RESPONSE {
table={}
table["operation"]="size"
ret=HTTP:payload(table)
debug("Size: %d \n",ret)
}
```

Note:

- The actual amount collected may be limited by system constraints and the available data size.
- The size parameter refers to the HTTP proxy block size, which is subject to FortiADC hardware limitations.
- Must be called in HTTP_REQUEST or HTTP_RESPONSE events to enable data buffering for subsequent HTTP_DATA_REQUEST or HTTP_DATA_RESPONSE events.

.

Supported Version

FortiADC version 4.8.x and later.

HTTP:payload(t)

Manipulates or inspects the buffered HTTP request or response body. This function is a unified interface for performing various operations on the payload data, such as retrieving its size or content, searching, modifying, replacing, or removing parts of it.

Syntax

HTTP:payload(t)

Arguments

Name	Description
t	A table that specifies the operation and its parameters. The table must contain an operation field. Other fields are required or optional depending on the operation.

Events

Applicable in the following events:

- HTTP_DATA_REQUEST
- HTTP_DATA_RESPONSE

Operations

The following sections detail all possible structures for the argument table **t**.

1. size – Returns the total size (in bytes) of the buffered payload.
2. content – Returns all or part of the buffered payload content as a string.
3. set – Overwrites or inserts data into the buffered payload at a specified location.
4. find – Searches for a string or regular expression within the buffered payload.
5. remove – Removes a string or regular expression from the buffered payload.
6. replace – Replaces all occurrences of a string or regular expression with a new string.

size

Returns the total size (in bytes) of the buffered payload.

Field	Type	Required	Description
operation	String	Yes	Must be set to "size".

Example

```
when HTTP_DATA_RESPONSE {
t1={}
t1["operation"]="size"
sz=HTTP:payload(t1)
debug("----response data size: %d-----\n", sz)}
```

content

Returns all or part of the buffered payload content as a string.

Field	Type	Required	Description
operation	String	Yes	Must be set to "content".
offset	Integer	No	The byte offset to start reading from. Default is 0.
size	Integer	No	The number of bytes to read. If missing, returns all data from the offset to the end.

Example

```
when HTTP_DATA_REQUEST {
t={};
t[“operation”]=”content”;   --return the buffered content
t[“offset”]=12;
t[“size”]=20;
ct = HTTP:payload(t);   --return value is a string containing the buffered content
}
```

Note: The “offset” and “size” fields are optional. If the “offset” field is missing, zero is assumed. If the “size” field is missing, it will operate on the whole buffered data.

set

Overwrites or inserts data into the buffered payload at a specified location.

Field	Type	Required	Description
operation	String	Yes	Must be set to "set".
offset	Integer	No	The byte offset where the data will be inserted/replaced. Default is 0.
size	Integer	No	The number of existing bytes to be overwritten starting from the offset. If missing, the operation will overwrite from the offset to the end of the buffer with the new data.
data	String	Yes	The new data string to insert.

Example

```
when HTTP_DATA_REQUEST {
t={};
t[“operation”]=”set” --replace the buffered content by new data
t[“offset”]=12;
t[“size”]=20;
t[“data”]= ”new data to insert”;
ret = HTTP:payload(t); --return value is boolean: false if fail, true if succeed
}
```

Note: The “offset” and “size” fields are optional. If the “offset” field is missing, zero is assumed. If the “size” field is missing, it will operate on the whole buffered data.

find

Searches for a string or regular expression within the buffered payload.

Field	Type	Required	Description
operation	String	Yes	Must be set to "find".
data	String	Yes	The string or regular expression pattern to search for (e.g., "text" or "t[ex]t").
offset	Integer	No	The byte offset to start searching from. Default is 0.
size	Integer	No	The number of bytes to search within. If missing, searches until the end of the buffer.
scope	String	No	Can be "first" (finds the first occurrence) or "all" (finds all occurrences). Default is "first".

Example

```
when HTTP_DATA_REQUEST {
t={};
t[“operation”]=”find”
t[“data”]=”sth”; -- can be a regular expression, like (s.h)
t[“offset”]=12;
t[“size”]=20;
t[“scope”]=”first” -- the scope field can be either “first” or “all”
ct = HTTP:payload(t);  --return value is a boolean false if operation fail or the number of
occurrences found;
}
```

Note: The “offset” and “size” fields are optional. If the “offset” field is missing, zero is assumed. If the “size” field is missing, it will operate on the whole buffered data. The “scope” field can be either be “first” or “all”.

remove

Removes a string or regular expression from the buffered payload.

Field	Type	Required	Description
operation	String	Yes	Must be set to "remove".
data	String	Yes	The string or regular expression pattern to remove.
offset	Integer	No	The byte offset to start the operation from. Default is 0.
size	Integer	No	The number of bytes to operate within. If missing, operates on the entire buffer from the offset.
scope	String	No	Can be "first" (remove the first match) or "all" (remove all matches). Default is "first".

Example

```
when HTTP_DATA_REQUEST {
t={};
t[“operation”]=”remove”
t[“data”]=”sth”; -- can be a regular expression, like (s.h)
t[“offset”]=12;
t[“size”]=20;
t[“scope”]=”first” --"first" or “all”
ct = HTTP:payload(t);  --return value is a boolean false if operation fail or the number of
occurrences removed
}
```

replace

Replaces all occurrences of a string or regular expression with a new string.

Field	Type	Required	Description
operation	String	Yes	Must be set to "replace".
data	String	Yes	The string or regular expression pattern to find.
new_data	String	Yes	The string to replace each match with.
offset	Integer	No	The byte offset to start the operation from. Default is 0.
size	Integer	No	The number of bytes to operate within. If missing, operates on the entire buffer from the offset.
scope	String	No	Can be "first" (remove the first match) or "all" (remove all matches). Default is "first".

Example

```
when HTTP_DATA_REQUEST {
t={};
t[“operation”]=”replace”
t[“data”]=”sth”; -- can be a regular expression, like (s.h)
t[“new_data”]=”sth new”; --“new_data” field is needed for the “replace” operation.
t[“offset”]=12;
t[“size”]=20;
t[“scope”]=”first” -- or “all”
ct = HTTP:payload(t);  --return value is a boolean false if operation fail or the number of
occurrences replaced
}
```

HTTP Cookie commands

Part of the HTTP class, Cookie commands manipulate cookie operation:

- HTTP:cookie_list() on page 283 – Returns a list of cookies: their names and values.
- HTTP:cookie(t) on page 284 – Allows you to get/set cookie value and cookie attribute, remove a whole cookie, get the whole cookie in HTTP_RESPONSE, and insert a new cookie.
- HTTP:cookie_crypto(t) on page 286 – The provided function response_encrypt_cookie can be used to perform cookie encryption in HTTP_RESPONSE and request_decrypt_cookie can be used to perform cookie decryption in HTTP_REQUEST.

HTTP:cookie_list()

Returns a list of cookies: their names and values.

Syntax

HTTP:cookie_list();

Arguments

N/A

Events

Applicable in the following events:

- `PERSISTENCE`
- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
ret=HTTP:cookie_list()
for k, v in pairs(ret) do
debug("cookie name %s, value %s\n", k, v);
end
}
```

Supported Version

FortiADC version 5.0.x and later.

HTTP:cookie(t)

Allows you to get/set cookie value and cookie attribute, remove a whole cookie, get the whole cookie in HTTP_RESPONSE, and insert a new cookie.

Syntax

HTTP:cookie(t);

t = {}

t[“name”] = “name”

t[“parameter”] = {can be value, cookie, path, domain, expires, secure, maxage, max-age, httponly, version, port, attrname }

t[“value”] = value

t[“case_sensitive”] = {0 or 1}

t[“action”] = {can be get, set, remove, insert}

ret = HTTP:cookie(t) --return is true if succeed or false if failed

Arguments

Name	Description
t	A table which specifies cookie name, parameter, action.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
t={};
t["name"]="test"
t["parameter"]="value";--value, cookie, path, domain, expires,  secure, maxage, max-age,
httponly,  version, port, attrname,
t["action"]="get"--get, set, remove, insert
ret = HTTP:cookie(t)
if ret then
debug("get cookie value succeed %s\n",ret);
else
debug("get cookie value failed\n");
end
}
```

Note:

- name: specify cookie name
- parameter: specify cookie value and attribute, including value, cookie, path, domain, expires, secure, maxage, max-age, httponly, version, port
- action: can be get, set, remove, insert

Supported Version

FortiADC version 5.0.x and later.

HTTP:cookie_crypto(t)

The provided function response_encrypt_cookie can be used to perform cookie encryption in HTTP_RESPONSE and request_decrypt_cookie can be used to perform cookie decryption in HTTP_REQUEST.

Syntax

HTTP:cookie_crypto(t);

Arguments

Name	Description
t	A table which specifies cookie name, parameter, action.

Events

Applicable in the following events:

- `BEFORE_AUTH`
- `HTTP_REQUEST`
- `HTTP_RESPONSE`
- `WAF_REQUEST_ATTACK_DETECTED`
- `WAF_REQUEST_BEFORE_SCAN`
- `WAF_RESPONSE_ATTACK_DETECTED`
- `WAF_RESPONSE_BEFORE_SCAN`

Example

```
when HTTP_REQUEST {
--encrypt cookie "test" in HTTP REQUEST before forwarding to real servers
local t={};
t["name"]="cookiename"
t["action"]="encrypt"      --encrypt, or decrypt
t["key"]="0123456789ABCDEF";
t["prefix"]="XXXX";
t["size"]=size-- 128, 192, or 256, the corresponding key length is 16, 24, and 32
if  HTTP:cookie_crypto(t) then
debug("Encrypt cookie succeed\n");
else
debug("Encrypt cookie failed\n");
end
}
```

Note:

- name: specify cookie name
- action: can be encrypt, or decrypt
- size: can be 128, 192, or 256, the corresponding key length is 16, 24, and 32

Supported Version

FortiADC version 5.2.x and later.

HTTP RAM cache commands

Before you begin, ensure RAM caching configuration is selected in the HTTP or HTTPs profile.

- HTTP:exclude_check_disable() on page 289 – Disables the exclude URI check.
- HTTP:cache_disable() on page 290 – Disables caching (both cache hit and cache store).
- HTTP:dyn_check_disable() on page 291 – Disables dynamic caching check.
- HTTP:dyn_invalid_check_disable() on page 292 – Disables dynamic invalid caching check.
- HTTP:dyn_cache_enable(t) on page 293 – Directly enables dynamic caching with a given ID and age.
- HTTP:cache_user_key(t) on page 294 – Replaces the default key (the URI) with any customized key.
- HTTP:dyn_cache_invalid(t) on page 295 – Invalidates a given dynamic cache indexed by its ID.
- HTTP:cached_check(t) on page 296 – Checks whether a URI has been cached or not.
- HTTP:cache_hits(t) on page 297 – Checks the cache hit count for the specified URI.
- HTTP:res_caching() on page 298 – Checks whether or not the response has been caching. If yes, then it checks whether it is regular cache or dynamic cache.
- HTTP:res_cached() on page 299 – Check whether or not a response is from cache. If yes, then it checks whether it is regular cache or dynamic cache.
- HTTP:caching_drop() on page 300 – Drops an ongoing caching operation.

HTTP:exclude_check_disable()

Disables the exclude URI check.

Syntax

HTTP:exclude_check_disable();

Arguments

N/A

Events

Applicable in HTTP_REQUEST.

Example

```
when HTTP_REQUEST {
if HTTP:exclude_check_disable() then
debug("set HTTP:exclude_check_disable: true\n");
else
debug("set HTTP:exclude_check_disable: Fail");
end
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:cache_disable()

Disables caching (both cache hit and cache store).

Syntax

HTTP: cache_disable();

Arguments

N/A

Events

`HTTP_REQUEST`

Example

```
when HTTP_REQUEST {
HTTP:cache_disable()
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:dyn_check_disable()

Disables dynamic caching check.

Syntax

HTTP: dyn_check_disable();

Arguments

N/A

Events

Applicable in HTTP_REQUEST.

Example

```
when HTTP_REQUEST {
HTTP:dyn_check_disable()
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:dyn_invalid_check_disable()

Disables dynamic invalid caching check.

Syntax

HTTP:dyn_invalid_check_disable();

Arguments

N/A

Events

Applicable in HTTP_REQUEST.

Example

```
when HTTP_REQUEST {
HTTP:dyn_invalid_check_disable()
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:dyn_cache_enable(t)

Directly enables dynamic caching with a given ID and age.

Syntax

HTTP:dyn_cache_enable(t);

Arguments

Name	Description
t	A table which specifies the ID and age of caching.

Events

Applicable in HTTP_REQUEST.

Example

```
when HTTP_REQUEST {
t={}
t["id"] = 1;
t["age"] = 20;     --in seconds
ret=HTTP:dyn_cache_enable(t)
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:cache_user_key(t)

Replaces the default key (the URI) with any customized key.

Syntax

HTTP:cache_user_key(t);

Arguments

Name	Description
t	A table which specifies the caching URI.

Events

HTTP_REQUEST.

Example

```
when HTTP_REQUEST {
url = HTTP:uri_get()
new_url = url.."external";
t={};
t["uri"] = new_url
HTTP:cache_user_key(t)
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:dyn_cache_invalid(t)

Invalidates a given dynamic cache indexed by its ID.

Syntax

HTTP:dyn_cache_invalid(t);

Arguments

Name	Description
t	A table which specifies the cache ID.

Events

Applicable in the following events:

- HTTP_REQUEST
- HTTP_RESPONSE

Example

```
when HTTP_REQUEST {
t={}
t["id"] = 1      --between 1 and 1023
ret = HTTP:dyn_cache_invalid(t);
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:cached_check(t)

Checks whether a URI has been cached or not.

Syntax

HTTP:cached_check(t);

Arguments

Name	Description
t	A table which specifies the cached URI.

Events

Applicable in the following events:

- HTTP_REQUEST
- HTTP_RESPONSE

Example

```
when HTTP_REQUEST {
t={}
t["uri"] = "/3.htm";
ret=HTTP:cached_check(t)
if ret then
debug("cached with id %s\n", ret);
else
debug("not cached\n");
end
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:cache_hits(t)

Checks the cache hit count for the specified URI.

Syntax

HTTP:cache_hits(t);

Arguments

Name	Description
t	A table which specifies the cached URI.

Events

```
Applicable in the following events:
```

- `HTTP_REQUEST`
- `HTTP_RESPONSE`

Example

```
when HTTP_REQUEST {
t={}
t["uri"] = "/3.htm";
ret=HTTP:cache_hits(t)
if ret then
debug("cache hit count %s\n", ret);
else
debug("not cached\n");
end
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:res_caching()

Checks whether or not the response has been caching. If yes, then it checks whether it is regular cache or dynamic cache.

Syntax

HTTP:res_caching();

Arguments

N/A

Events

Applicable in HTTP_RESPONSE.

Example

```
when HTTP_RESPONSE {
id = HTTP:res_caching();
if  id then
debug("HTTP:res_caching() response caching with id %s !!!!\n", id);
else
debug("HTTP:res_caching() response NOT caching\n");
end
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:res_cached()

Check whether or not a response is from cache. If yes, then it checks whether it is regular cache or dynamic cache.

Syntax

HTTP:res_cached();

Arguments

N/A

Events

Applicable in HTTP_RESPONSE.

Example

```
when HTTP_RESPONSE {
ret = HTTP:res_cached();
if  ret then
debug("HTTP:res_cached() response from cache !!!!\n");
else
debug("HTTP:res_cached() response NOT from cache\n");
end
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP:caching_drop()

Drops an ongoing caching operation.

Syntax

HTTP:caching_drop();

Arguments

N/A

Events

Applicable in HTTP_RESPONSE.

Example

```
when HTTP_RESPONSE {
ret=HTTP:caching_drop()
if ret then
debug("script dropping caching: True\n")
else
debug("script dropping caching: False\n");
end
}
```

Supported Version

FortiADC version 5.3.x and later.

HTTP Persistence commands

- HTTP:persist(t) on page 302 – Manages persistence/stickiness table operations, including reading, writing, dumping table contents, and retrieving server information.
- HTTP:lookup_tbl(t) on page 308 – Use this hash value to lookup the stick table, and then persist the session.
- PROXY:init_stick_tbl_timeout(int) on page 310 – Sets the timeout of the stick table.

HTTP:persist(t)

Manages persistence/stickiness table operations, including reading, writing, dumping table contents, and retrieving server information.

Syntax

HTTP:persist(t);

Arguments

Name	Description
t	A table that specifies the operation and its parameters.

Events

Applicable in the following events:

- PERSISTENCE
- POST_PERSIST
- HTTP_REQUEST (supported since version 7.2.x)
- HTTP_DATA_REQUEST, HTTP_DATA_RESPONSE (supported since version 8.0.0)

Operations

The following sections detail all possible structures for the argument table **t**.

1. save_tbl – Saves a server assignment for a hash value in the persistence table.
2. read_tbl – Saves a server assignment for a hash value in the persistence table.
3. dump_tbl – Dumps a range of entries from the persistence table.
4. get_valid_server – Saves a server assignment for a hash value in the persistence table.
5. cal_server_from_hash – Saves a server assignment for a hash value in the persistence table.
6. get_current_assigned_server – Saves a server assignment for a hash value in the persistence table.

save_tbl

Saves a server assignment for a hash value in the persistence table.

Field	Type	Required	Description
operation	String	Yes	Must be "save_tbl".
hash_value	String	Yes	The hash key to store.
srv_name	Integer	String	The server name to associate with the hash.

Example

```
when PERSISTENCE {
    cip = HTTP:client_addr()
    hash_str_cip = sha512_hex(cip)
    debug("-----save_tbl-----\n")
    t={}
    t["operation"] = "save_tbl"
    t["hash_value"] = hash_str_cip
    t["srv_name"] = "pool1-2"
    ret = HTTP:persist(t)
    if ret then
           debug("hash save table success\n");
       else
           debug("save table failed\n");
       end
       t={};
       t["operation"] = "save_tbl";
       t["hash_value"] = "246810";
       t["srv_name"] = "pool1-3";
       ret = HTTP: persist(t);
       if ret then
           debug("===server add success\n");
    else
        debug("===server add fail\n");
    end
}
Output:
true: success, false: failed
```

Note: Due to limitations in the stick table, it only supports 16 characters in the hash value. Otherwise, we will hash it to obtain 16 bytes as index.

Supported Version

FortiADC version 5.4.x and later. In 7.2.x, function extended to HTTP_REQUEST events.

read_tbl

Saves a server assignment for a hash value in the persistence table.

Field	Type	Required	Description
operation	String	Yes	Must be "read_tbl".
hash_value	String	Yes	The hash key to lookup

Example

```
when PERSISTENCE {
       t={};
       t["operation"] = "save_tbl";
       t["hash_value"] = "246810";
       t["srv_name"] = "pool1-3";
       ret = HTTP: persist(t);
       if ret then
           debug("===server add success\n");
    else
        debug("===server add fail\n");
    end

    t={}
    t["operation"] = "read_tbl"
    t["hash_value"] = "246810"
    ret_tbl = HTTP:persist(t)
    if ret_tbl then
        debug("246810-server: %s\n",ret_tbl)
    else
        debug("246810-server read fail\n")
end
}
Output:
       Return server name of the entry, or false if no entry found
```

Supported Version

FortiADC version 5.4.x and later. In 7.2.x, function extended to HTTP_REQUEST events.

dump_tbl

Dumps a range of entries from the persistence table.

Field	Type	Required	Description
operation	String	Yes	Must be "dump_tbl".
index	Integer	No	Starting index for dump (default: 1).
count	Integer	No	Number of entries to return (default: all).

Example

```
when PERSISTENCE {
       t={};
       t["operation"] = "save_tbl";
       t["hash_value"] = "246810";
       t["srv_name"] = "pool1-3";
```

```
        ret = HTTP: persist(t);
        if ret then
            debug("===server add success\n");
    else
        debug("===server add fail\n");
    end

    t={}
    t["operation"] = "dump_tbl"
    t[“index”] = 1
    t[“count”] = 15
    ret_tbl = HTTP:persist(t)
    if ret_tbl then
        for hash, srv in pairs(ret_tbl) do
            debug("tbl hash %s srv %s\n",hash,srv)
        end
    end
}
       “index”:
      “count”:
Output:
       Return A table include hash and server name
```

Supported Version

FortiADC version 5.4.x and later. In 7.2.x, function extended to HTTP_REQUEST events.

get_valid_server

Saves a server assignment for a hash value in the persistence table.

Field	Type	Required	Description
operation	String	Yes	Must be "get_valid_server".

Example

```
when PERSISTENCE {
    debug("-----get valid server-----\n")
    t={}
    t["operation"] = "get_valid_server"
    ret = HTTP:persist(t)
    if ret then
        for srv,stat in pairs(ret) do
            debug("server %s, status %s\n",srv,stat)
        end
    end
}
Output:
       Return the table of usable real server and server state(enable, backup)
```

Supported Version

FortiADC version 5.4.x and later. In 7.2.x, function extended to HTTP_REQUEST events.

cal_server_from_hash

Saves a server assignment for a hash value in the persistence table.

Field	Type	Required	Description
operation	String	Yes	Must be "cal_server_from_hash".
hash_value	String	Yes	The hash value to calculate from.

Example

```
when PERSISTENCE {
    debug("-----cal_server_from_hash-----\n")
    t={}
    t["operation"] = "cal_server_from_hash"
    t["hash_value"] = "246810"
    ret = HTTP:persist(t)
    if ret then
        debug("hash 246810, server %s\n",ret)
    end
}
Output:
       Return the real server name according to the hash value using our algorithm or False if failed
```

Supported Version

FortiADC version 5.4.x and later. In 7.2.x, function extended to HTTP_REQUEST events.

get_current_assigned_server

Saves a server assignment for a hash value in the persistence table.

Field	Type	Required	Description
operation	String	Yes	Must be "get_current_assigned_server".

Example

```
when PERSISTENCE {
    cip = HTTP:client_addr()
    hash_str_cip = sha512_hex(cip)
    t={}
    t["operation"] = "save_tbl"
    t["hash_value"] = hash_str_cip
    t["srv_name"] = "pool1-3"
    ret = HTTP:persist(t)
    if ret then
           debug("hash save table success\n");
       else
           debug("save table failed\n");
       end
    t={}
    t["hash_value"]=hash_str_cip
    ret = HTTP:lookup_tbl(t)
    if ret then
        debug("hash LOOKUP success\n")
    else
        debug("hash lookup failed\n")
    end
}
when POST_PERSIST {
debug("-----event POST_PERSIST-----\n")
    debug("-----get current assigned server-----\n")
    t={}
    t["operation"]="get_current_assigned_server"
    ret=HTTP:persist(t)
    debug("current assigned server: %s\n",ret)
}
Output:
       Return the real server name which is assigned to current session or False if no server is
assigned right now
```

Supported Version

FortiADC version 5.4.x and later. In 7.2.x, function extended to HTTP_REQUEST events.

HTTP:lookup_tbl(t)

Use this hash value to lookup the stick table, and then persist the session.

Syntax

HTTP:lookup_tbl(t);

Arguments

Name	Description
t	A table specifies the operation and hash value.

Events

Applicable in the following events:

- PERSISTENCE
- HTTP_REQUEST
- HTTP_DATA_REQUEST (supported since version 8.0.0)

Example

```
when PERSISTENCE {
        cip = HTTP:client_addr()
    hash_str_cip = sha512_hex(cip)

    t={}
    t["operation"] = "save_tbl"
    t["hash_value"] = hash_str_cip
    t["srv_name"] = "pool1-3"
    ret = HTTP:persist(t)
    if ret then
           debug("hash save table success\n");
       else
           debug("save table failed\n");
       end

    t={}
    t["hash_value"]=hash_str_cip
    ret = HTTP:lookup_tbl(t)
    if ret then
        debug("hash LOOKUP success\n")
    else
```

```
        debug("hash lookup failed\n")
    end
}
Output:
       Return True: lookup success, False, lookup failed and use the org. LB method
```

Supported Version

FortiADC version 5.4.x and later.

PROXY:init_stick_tbl_timeout(int)

Sets the timeout of the stick table.

Syntax

PROXY:init_stick_tbl_timeout(int);

Arguments

Name	Description
int	A positive integer that specifies the timeout.

Events

Applicable in RULE_INIT.

Example

```
when RULE_INIT {
    env={}
    PROXY:init_stick_tbl_timeout(500)
}
when PERSISTENCE {
    cip = HTTP:client_addr()
    hash_str_cip = sha512_hex(cip)

    debug("-----save_tbl-----\n")

    t={}
    t["operation"] = "save_tbl"
    t["hash_value"] = hash_str_cip
    t["srv_name"] = "pool1-3"

    ret = HTTP:persist(t)

    if ret then
        debug("hash save table success\n");
    else
        debug("save table failed\n");
    end
}
```

```
Output:
      Return True: success, False: failed
```

Supported Version

FortiADC version 5.4.x and later.

LB commands

LB commands contain load balancing manipulating functions, with the most important function being LB:routing which allows you to select the backend:

- LB:routing(value) on page 313 – Routes the request to the content routing server.
- LB:get_valid_routing() on page 314 – Returns a list of backend names configured on the current Virtual Server in the form of a Lua table (number as key, string as value).
- LB:get_current_routing() on page 315 – Returns the currently allocated backend for this request.
- LB:method_assign_server() on page 316 – Returns the server through the current load balance method configured on current the Virtual Server.
- LB:set_real_server(rs_name) on page 317 – Sets a real server from the configured pool directly.

LB:routing(value)

Routes the request to the content routing server.

Syntax

LB:routing(value);

Arguments

Name	Description
value	A string which specifies the content routing to route.

Events

Applicable in the following events:

- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`

Example

```
when HTTP_REQUEST {
LB:routing(“content_routing1”);
}
--supports multiple routing
LB:routing(“cr1”)
LB:routing(“cr2”)
--It will be routed to cr2; the final one prevails.
```

Note:

When a VS enables both content-routing and scripting, then this function will perform a cross-check to check the content-routing used in scripting is applied to the VS.

Supported Version

FortiADC version 4.3.x and later.

LB:get_valid_routing()

Returns a list of backend names configured on the current Virtual Server in the form of a Lua table (number as key, string as value).

Since there must be at least one backend, the returned value cannot be nil or empty. In case of a single routing with zero or one content routing configured, the pool name will be returned.

Syntax

LB:get_valid_routing();

Arguments

N/A

Events

Applicable in the following events:

- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`

Example

```
when HTTP_REQUEST {
t=LB:get_valid_routing()
for k, v in pairs(t) do
debug("Key: %d Value: %s\n", k, v)
end
}
```

Supported Version

FortiADC version 7.4.2 and later.

LB:get_current_routing()

Returns the currently allocated backend for this request.

If there is only one backend configured, this function returns the backend name in a string format. If there are multiple backends available, and they are already set via LB:routing(), then this function returns the backend name as a string. Otherwise, this returns an empty string if no backend is configured.

In case of a single routing with zero or one content routing configured, the pool name will be returned.

Syntax

LB:get_current_routing();

Arguments

N/A

Events

Applicable in the following events:

- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`

Example

```
when HTTP_REQUEST {
    s=LB:get_current_routing()
    if (s == nil or s == '') then
        s = "No backend returned."
    end
    debug("current routing: %s\n", s)
}
```

Supported Version

FortiADC version 7.4.2 and later.

LB:method_assign_server()

Returns the server through the current load balance method configured on current the Virtual Server.

The implementation will first check whether the backend is already set. If the backend is set, then it runs the load balance algorithm to assign a server and returns the server name as a string. If the backend is not set, the function returns an empty string, and then FortiADC will set the backend only if there is only one backend available. If there is no server available in the corresponding pool or all the servers are down in the pool, an empty string will also be returned.

Syntax

LB:method_assign_server();

Arguments

N/A

Events

Applicable in the following events:

- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`

Example

```
when HTTP_REQUEST {
    s=LB:method_assign_server()
    if (s == nil or s == '') then
    s = "No Server returned."
    end
   debug("assign server: '%s'\n", s)
}
```

Note:

The result may be overwritten by functions in PERSISTENCE events, which happen after HTTP_REQUEST. For example, this function returns server01. But the PERSISTENCE event specifies to use server02. The result from the PERSISTENCE event (server02) will always supercede results from non-PERSISTENCE events (server01).

Supported Version

FortiADC version 7.4.2 and later.

LB:set_real_server(rs_name)

Sets a real server from the configured pool directly.

This function returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

LB:set_real_server(rs_name);

Arguments

Name	Description
rs_name	A Lua string as the name of the real server. A valid real server must meet the following conditions: • The real server name is valid and complies with name requirements (character limit, etc.) • The real server's is a member of the selected pool. • The real server is usable.

Events

Applicable in the following events:

- `AUTH_RESULT`
- `BEFORE_AUTH`
- `HTTP_DATA_REQUEST`
- `HTTP_REQUEST`
- `PERSISTENCE`
- `POST_PERSIST`

Examples

For events with manual routing selection:

```
when HTTP_REQUEST {
       debug("===>>===========begin HTTP scripting===========\n")
       -- routing to set
       sr = "test07"
       s = LB:get_current_routing()
       if (s == nil or s == "") then
           debug("===>>set routing to be '%s'\n", sr)
               LB:routing(sr)
```

```
                s = sr
        end
        debug("===>>current routing: %s\n", s)

     -- Real Server name
        rs = "rs05"
        ret = LB:set_real_server(rs)
        if ret then
                debug("LB:set_real_server(%s) Success.\n", rs);
        else
         debug("LB:set_real_server(%s) Failed.\n", rs);
        end
        debug("===>>============end  HTTP scripting===========\n")
}
```

For events without manual routing selection:

```
when PERSISTENCE {
        debug("===>>===========begin scripting===========\n")
        -- Real Server name
        rs = "rs06"
        ret = LB:set_real_server(rs)
        if ret then
                debug("set_real_server(%s) Success.\n", rs);
        else
         debug("set_real_server(%s) Failed.\n", rs);
        end
        debug("===>>============end scripting===========\n")
}
```

Supported Version

FortiADC version 7.4.3 and later.

PROXY commands

- PROXY:set_auth_key(value) on page 321 – Customize the crypto key FortiADC used for encrypt/decrypt authentication cookie name "FortiADCauthSI". This will increase your FortiADC's security so that others cannot forge this authentication cookie.
- PROXY:clear_auth_key(value) on page 322 – Clears the customized authentication key that was previously set to use the default key instead.
- PROXY:shared_table_create(table_name,[entry_size],[memory_limit]) on page 323 – Creates a shared table if there is no existing shared table with the specified name.
- PROXY:shared_table_destroy(table_name) on page 325 – Destroys the specified shared table and all the data entries if the calling process is the only one attached to the shared table. In case there is more than one process attached to this table, this function will only detach both the data entries and the shared table for the calling process.
- PROXY:shared_table_entry_count(table_name) on page 326 – Returns the current count of entries in a shared table. Returns -1 if the table does not exist.
- PROXY:shared_table_memory_size(table_name) on page 327 – Returns the current memory usage of a shared table. Returns -1 if the table does not exist.
- PROXY:shared_table_insert(table_name, key, value) on page 328 – Inserts a pair of <key, value> as an entry into the shared table. If the key already exists in the table or if the table is full, the function will do nothing. The key can be a Lua string or integer while the value can be a Lua string, integer, or table.
- PROXY:shared_table_lookup(table_name, key) on page 330 – Looks up whether a key exists in the shared table. If the key exists, returns the corresponding value.
- PROXY:shared_table_delete(table_name, key) on page 331 – Deletes an entry specified by a key from the shared table. If the key does not exist, the function will do nothing. If there is more than one process attached to the data entry, this function only detaches the calling process.
- PROXY:shared_table_dump(table_name, [index], [count]) on page 332 – Prints the current contents of the shared table for debugging purposes. This works similar to an iterator for a shared table.
- PROXY:atomic_counter_create(counter_name) on page 334 – Creates an atomic counter with a specific name. This counter is sharable among different processes of one virtual server. The creation process has the ownership. All the other processes within the same VS can perform all the operations except destroy. Each VS can create up to 255 counters.
- PROXY:atomic_counter_destroy(counter_name) on page 335 – Destroys an atomic counter with a specific name. The creator process has the ownership, and only the owner can destroy the counter. Non-owner calls will be ignored.
- PROXY:atomic_counter_set(counter_name, value) on page 336 – Sets the value specified by an atomic counter name.
- PROXY:atomic_counter_get(counter_name) on page 337 – Returns the current value of the specified atomic counter name.
- PROXY:atomic_counter_inc(counter_name) on page 338 – Increases the current value of the specified atomic counter name by one.
- PROXY:atomic_counter_dec(counter_name) on page 339 – Decreases the current value of the specified atomic counter name by one.
- PROXY:atomic_counter_add(counter_name,value) on page 340 – Adds a value to the specified atomic counter name.

- PROXY:atomic_counter_sub(counter_name, value) on page 341 – Subtracts a value from the specified atomic counter name.

PROXY:set_auth_key(value)

Customize the crypto key FortiADC used for encrypt/decrypt authentication cookie name "FortiADCauthSI". This will increase your FortiADC's security so that others cannot forge this authentication cookie.

Syntax

PROXY:set_auth_key(value);

Arguments

Name	Description
value	A string which will be used to encrypt/decrypt the authentication cookie. Value length is fixed, must be 32 bytes.

Events

Applicable in the following events:

- VS_LISTENER_BIND

Example

```
when VS_LISTENER_BIND {
    AUTH_KEY = “0123456789ABCDEF0123456789ABCDEF”
    result = PROXY:set_auth_key(AUTH_KEY)
    If result then
        debug(“set auth key succeed\n”)
     end
}
```

Supported Version

FortiADC version 5.2.x and later.

PROXY:clear_auth_key(value)

Clears the customized authentication key that was previously set to use the default key instead.

Syntax

PROXY:clear_auth_key(value);

Arguments

Name	Description
value	A string which will be used to encrypt/decrypt the authentication cookie.

Events

Applicable in the following events:

- VS_LISTENER_BIND

Example

```
when TCP_BIND {
result = PROXY:clear_auth_key("newkey")
}
```

Supported Version

FortiADC version 5.2.x and later.

PROXY:shared_table_create(table_name,[entry_size],[memory_limit])

Creates a shared table if there is no existing shared table with the specified name. Multiple shared tables can be created with different names. Each table operation must specify a table name.

Before creating the table, FortiADC will check whether there is already a table with the same name. If a table with the same name already exists, it will either be attached or will return an error. Typically, the first calling process creates the table, then all the other processes will attach to it with the same API call.

For any table, this API must be called first before other operations can be performed.

Returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

PROXY:shared_table_create(table_name, [entry_size], [memory_limit]);

Arguments

Name	Description
table_name	A Lua string as the name of the shared table. This is the unique identification of a shared table. This parameter is mandatory. The maximum length of this table name is 255.
entry_size	The maximum number of entries this table can hold. This parameter is optional. The default value is 2048. Must not exceed 2048.
memory_limit	The maximum amount of memory that can be allocated for a shared table and its data entries. This parameter is optional. The default value is 4 G.

Events

All events except: PERSISTENCE, POST_PERSIST, CLIENTSSL_HANDSHAKE

Example

```
when HTTP_REQUEST {
       table_name = "TableDemo1"
       ret = PROXY:shared_table_create(table_name, 2048, 20971520)
       if ret then
          debug("===>>shared_table_create success: [%s]\n", table_name)
       else
          debug("===>>shared_table_create failed: [%s]\n", table_name)
       end
}
```

Supported Version

FortiADC version 7.4.2 and later.

PROXY:shared_table_destroy(table_name)

Destroys the specified shared table and all the data entries if the calling process is the only one attached to the shared table. In case there is more than one process attached to this table, this function will only detach both the data entries and the shared table for the calling process.
Returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

PROXY:shared_table_destroy(table_name);

Arguments

Name	Description
table_name	A Lua string as the name of the shared table. This is the unique identification of a shared table. This parameter is mandatory. The maximum length of this table name is 255.

Events

All events except: PERSISTENCE, POST_PERSIST, CLIENTSSL_HANDSHAKE.

Example

```
when HTTP_REQUEST {
       table_name = "TableDemo1"
       ret = PROXY:shared_table_destroy(table_name)
       if ret then
          debug("===>>shared_table_destroy success: [%s]\n", table_name)
       else
          debug("===>>shared_table_destroy failed: [%s]\n", table_name)
       end
}
```

Supported Version

FortiADC version 7.4.2 and later.

PROXY:shared_table_entry_count(table_name)

Returns the current count of entries in a shared table. Returns -1 if the table does not exist.

Syntax

PROXY:shared_table_entry_count(table_name);

Arguments

Name	Description
table_name	A Lua string as the name of the shared table. This is the unique identification of a shared table. This parameter is mandatory. The maximum length of this table name is 255.

Events

All events except: PERSISTENCE, POST_PERSIST, CLIENTSSL_HANDSHAKE.

Example

```
when HTTP_REQUEST {
	table_name = "TableDemo1"
	ret = PROXY:shared_table_entry_count(table_name)
	debug("===>>shared_table_entry_count: [%s]=[%d]\n", table_name, ret)
}
```

Supported Version

FortiADC version 7.4.2 and later.

PROXY:shared_table_memory_size(table_name)

Returns the current memory usage of a shared table. Returns -1 if the table does not exist.

Syntax

PROXY:shared_table_memory_size(table_name);

Arguments

Name	Description
table_name	A Lua string as the name of the shared table. This is the unique identification of a shared table. This parameter is mandatory. The maximum length of this table name is 255.

Events

All events except: PERSISTENCE, POST_PERSIST, CLIENTSSL_HANDSHAKE.

Example

```
when HTTP_REQUEST {
       table_name = "TableDemo1"
       ret = PROXY:shared_table_memory_size(table_name)
       debug("===>>shared_table_memory_size: [%s]=[%d]\n", table_name, ret)
}
```

Supported Version

FortiADC version 7.4.2 and later.

PROXY:shared_table_insert(table_name, key, value)

Inserts a pair of <key, value> as an entry into the shared table. If the key already exists in the table or if the table is full, the function will do nothing. The key can be a Lua string or integer while the value can be a Lua string, integer, or table.

This function will fail if the table has not been created yet; it will not trigger the creation of a new table.

Returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

PROXY:shared_table_insert(table_name, key, value);

Arguments

Name	Description
table_name	A Lua string as the name of the shared table. This is the unique identification of a shared table. This parameter is mandatory. The maximum length of this table name is 255.
key	The key can be a Lua string or integer. This parameter is mandatory. The maximum length for the string is 255.
value	The value can be a Lua string, integer, or table. The table will be serialized before storing into the shared table. This parameter is mandatory. The maximum length of any value is 64K.

Events

All events except: PERSISTENCE, POST_PERSIST, CLIENTSSL_HANDSHAKE.

Example

```
when HTTP_REQUEST {
       table_name = "TableDemo1"
       key1="keyString101"
       val1="valString101"
       ret = PROXY:shared_table_insert(table_name, key1, val1)
       if ret then
          debug("===>>shared_table_insert success:[Table:%s] [%s]=[%s]\n",table_name, key1, val1)
       else
          debug("===>>shared_table_insert failed:[Table:%s] [%s]=[%s]\n",table_name, key1, val1)
       end
}
```

Supported Version

FortiADC version 7.4.2 and later.

PROXY:shared_table_lookup(table_name, key)

Looks up whether a key exists in the shared table. If the key exists, returns the corresponding value.

This function will fail if the table has not been created yet; it will not trigger the creation of a new table.

Returns the stored value if successful, otherwise, returns Boolean false. The returned value can be an Lua string, integer, or table. The table will be deserialized before returning, so the API will receive a Lua table.

Syntax

PROXY:shared_table_lookup(table_name, key);

Arguments

Name	Description
table_name	A Lua string as the name of the shared table. This is the unique identification of a shared table. This parameter is mandatory. The maximum length of this table name is 255.
key	The key can be a Lua string or integer. This parameter is mandatory. The maximum length for the string is 255.

Events

All events except: PERSISTENCE, POST_PERSIST, CLIENTSSL_HANDSHAKE.

Example

```
when HTTP_REQUEST {
       table_name = "TableDemo1"
       key1="keyString101"
       ret = PROXY:shared_table_lookup(table_name,key1)
       if ret then
          debug("===>>shared_table_lookup success: [Table:%s]  [%s]=[%s]\n", table_name, key1, ret)
       else
          debug("===>>shared_table_lookup failed for key: [Table:%s] [%s]\n", table_name, key1)
       end
}
```

Supported Version

FortiADC version 7.4.2 and later.

PROXY:shared_table_delete(table_name, key)

Deletes an entry specified by a key from the shared table. If the key does not exist, the function will do nothing. If there is more than one process attached to the data entry, this function only detaches the calling process.

This function will fail if the table has not been created yet; it will not trigger the creation of a new table.

Returns Boolean true if successful, otherwise, returns Boolean false.

Syntax

PROXY:shared_table_delete(table_name, key);

Arguments

Name	Description
table_name	A Lua string as the name of the shared table. This is the unique identification of a shared table. This parameter is mandatory. The maximum length of this table name is 255.
key	The key can be a Lua string or integer. This parameter is mandatory. The maximum length for the string is 255.

Events

All events except: PERSISTENCE, POST_PERSIST, CLIENTSSL_HANDSHAKE.

Example

```
when HTTP_REQUEST {
       table_name = "TableDemo1"
       key3 ="keyString103"
       ret = PROXY:shared_table_delete(table_name,key3)
       if ret then
          debug("===>>shared_table_delete success for key: [Table:%s]  [%s]\n", table_name, key3)
       else
          debug("===>>shared_table_delete failed for key: [Table:%s]  [%s]\n", table_name, key3)
       end
}
```

Supported Version

FortiADC version 7.4.2 and later.

PROXY:shared_table_dump(table_name, [index], [count])

Prints the current contents of the shared table for debugging purposes. This works similar to an iterator for a shared table.

Returns a pair table, which can be traversed with for [k, v]. All keys and values will be converted into strings. Returns NIL if there is any error or the table is empty.

Syntax

PROXY:shared_table_dump(table_name, [index], [count]);

Arguments

Name	Description
table_name	A Lua string as the name of the shared table. This is the unique identification of a shared table. This parameter is mandatory. The maximum length of this table name is 255.
index	A Lua integer is used as the print index. This will indicate the order of printing for all the items. This parameter is optional. The default value is 1. The valid range is from 1 to the current entry count of the table. FortiADC will check the validity of this index and report errors for invalid entries. **Note**: The item order in the hash table is not important, so there is no need to try to match the index to the item.
count	A Lua integer is used to indicate the number of items to print. This parameter is optional. The default value is 1000 or the current entry count (whichever is smaller). The valid range is from 1 to the current entry count of the table. FortiADC will check the validity of this count and report errors for invalid entries. The index is always processed before the count, so if only one parameter exists, then it is assumed as the index. The actual returned count may be less than the specified value if we run out of items or if there is any error.

Events

All events except: PERSISTENCE, POST_PERSIST, CLIENTSSL_HANDSHAKE.

Example

```
when HTTP_REQUEST {
       table_name = "TableDemo1"
       index = 1
       count = PROXY:shared_table_entry_count(table_name)
       ret = PROXY:shared_table_dump(table_name, index, count)
       --Or simply:
       --ret = PROXY:shared_table_dump(table_name)
       if ret then
          debug("===>>PROXY-shared_table_dump success [Table-%s]\n", table_name)
          for k, v in pairs(ret) do
             debug("===>>Key: %s Value: %s\n", k, v)
          end
       else
          debug("===>>PROXY-shared_table_dump failed [Table-%s]\n", table_name)
       end
}
```

Supported Version

FortiADC version 7.4.2 and later.

PROXY:atomic_counter_create(counter_name)

Creates an atomic counter with a specific name. This counter is sharable among different processes of one virtual server. The creator process has the ownership. All the other processes within the same VS can perform all the operations except destroy. Each VS can create up to 255 counters.

Syntax

PROXY:atomic_counter_create(counter_name);

Arguments

Name	Description
counter_name	A Lua string as the name of the counter. This parameter is mandatory.

Events

All events except: PERSISTENCE, POST_PERSIST.

Example

```
when HTTP_REQUEST {
          counter_name = "DemoCounter1"
       ret = PROXY:atomic_counter_create(counter_name)
       if ret then
          debug("===>>atomic_counter_create success:[%s]\n",counter_name)
       else
          debug("===>>atomic_counter_create failed:[%s]\n",counter_name)
       end
}
```

Supported Version

FortiADC version 7.6.0 and later.

PROXY:atomic_counter_destroy(counter_name)

Destroys an atomic counter with a specific name. The creator process has the ownership, and only the owner can destroy the counter. Non-owner calls will be ignored.

Syntax

PROXY:atomic_counter_destroy(counter_name)

Arguments

Name	Description
counter_name	A Lua string as the name of the counter. This parameter is mandatory.

Events

All events except: PERSISTENCE, POST_PERSIST.

Example

```
when HTTP_REQUEST {
         counter_name = "DemoCounter1"
         ret = PROXY:atomic_counter_destroy(counter_name)
         if ret then
                 debug("===>>atomic_counter_destroy success:[%s]\n",counter_name)
         else
                debug("===>>atomic_counter_destroy failed:[%s]\n",counter_name)
         end
}
```

Supported Version

FortiADC version 7.6.0 and later.

PROXY:atomic_counter_set(counter_name, value)

Sets the value specified by an atomic counter name.

Syntax

PROXY:atomic_counter_set(counter_name, value);

Arguments

Name	Description
counter_name	A Lua string as the name of the counter. This parameter is mandatory. This will map to an index of an array.
value	A Lua number to be assigned to the counter.

Events

All events except: PERSISTENCE, POST_PERSIST.

Example

```
when HTTP_REQUEST {
        counter_name = "DemoCounter1"
        value = 10
        ret = PROXY:atomic_counter_set(counter_name, value)
       if ret then
               debug("===>>atomic_counter_set success: [%s]=[%d]\n", counter_name, value)
       else
              debug("===>>atomic_counter_set failed: [%s]=[%d]\n", counter_name, value)
       end
}
```

Supported Version

FortiADC version 7.6.0 and later.

PROXY:atomic_counter_get(counter_name)

Returns the current value of the specified atomic counter name.

Syntax

PROXY:atomic_counter_get(counter_name);

Arguments

Name	Description
counter_name	A Lua string as the name of the counter. This parameter is mandatory. This will map to an index of an array.

Events

All events except: PERSISTENCE, POST_PERSIST.

Example

```
when HTTP_REQUEST {
       counter_name = "DemoCounter1"
       ret = PROXY:atomic_counter_get(counter_name)
       debug("===>>atomic_counter_get: [%s]=[%d]\n", counter_name, ret)
}
```

Supported Version

FortiADC version 7.6.0 and later.

PROXY:atomic_counter_inc(counter_name)

Increases the current value of the specified atomic counter name by one.

Syntax

PROXY:atomic_counter_inc(counter_name);

Arguments

Name	Description
counter_name	A Lua string as the name of the counter. This parameter is mandatory.

Events

All events except: PERSISTENCE, POST_PERSIST.

Example

```
when HTTP_REQUEST {
       counter_name = "DemoCounter1"
       ret = PROXY:atomic_counter_inc(counter_name)
       if ret then
            debug("===>>atomic_counter_inc success:[%s]\n",counter_name)
       else
            debug("===>>atomic_counter_inc failed:[%s]\n",counter_name)
       end
}
```

Supported Version

FortiADC version 7.6.0 and later.

PROXY:atomic_counter_dec(counter_name)

Decreases the current value of the specified atomic counter name by one.

Syntax

PROXY:atomic_counter_dec(counter_name);

Arguments

Name	Description
counter_name	A Lua string as the name of the counter. This parameter is mandatory.

Events

All events except: PERSISTENCE, POST_PERSIST.

Example

```
when HTTP_REQUEST {
       counter_name = "DemoCounter1"
       ret = PROXY:atomic_counter_dec(counter_name)
       if ret then
             debug("===>>atomic_counter_dec success: [%s]\n", counter_name)
       else
             debug("===>>atomic_counter_dec failed: [%s]\n", counter_name)
       end
}
```

Supported Version

FortiADC version 7.6.0 and later.

PROXY:atomic_counter_add(counter_name,value)

Adds a value to the specified atomic counter name.

Syntax

PROXY:atomic_counter_add(counter_name,value);

Arguments

Name	Description
counter_name	A Lua string as the name of the counter. This parameter is mandatory.
value	A Lua number to be added to the counter

Events

All events except: PERSISTENCE, POST_PERSIST.

Example

```
 when HTTP_REQUEST {
       counter_name = "DemoCounter1"
       value = 10
       ret = PROXY:atomic_counter_add(counter_name, value)
       if ret then
       debug("===>>atomic_counter_add success: [%s]=[%d]\n", counter_name, value)
       else
       debug("===>>atomic_counter_add failed: [%s]=[%d]\n", counter_name, value)
       end
}
```

Supported Version

FortiADC version 7.6.0 and later.

PROXY:atomic_counter_sub(counter_name, value)

Subtracts a value from the specified atomic counter name.

Syntax

PROXY:atomic_counter_sub(counter_name, value);

Arguments

Name	Description
counter_name	A Lua string as the name of the counter. This parameter is mandatory.
value	A Lua number to be subtracted from the counter. This cannot exceed the current counter value as the counter cannot be a negative number.

Events

All events except: PERSISTENCE, POST_PERSIST.

Example

```
when HTTP_REQUEST {
       counter_name = "DemoCounter1"
       value = 10
       ret = PROXY:atomic_counter_sub(counter_name, value)
       if ret then
       debug("===>>atomic_counter_sub success: [%s]=[%d]\n", counter_name, value)
       else
       debug("===>>atomic_counter_sub failed: [%s]=[%d]\n", counter_name, value)
       end
}
```

Supported Version

FortiADC version 7.6.0 and later.

WAF commands

WAF commands contain functions for obtaining and manipulating WAF related result information:

- WAF:enable() on page 343 – Enables the current session's WAF scan function.
- WAF:disable() on page 344 – Disables the current session's WAF scan function.
- WAF:status() on page 345 – Returns a status string to specify the current status of WAF detection. The status may be "enable" or "disable".
- WAF:action() on page 346 – Returns the current session's WAF action. This can only be called in an ATTACK_DETECTED event.
- WAF:override_action(action [, parameter]) on page 347 – Overrides the current stage's detected action to the specified.
- WAF:violations() on page 349 – Returns a table that includes all the violations detected by the current WAF stage as string values.
- WAF:abandon_violation(signature_id) on page 351 – Removes a violation by the specified signature ID. The signature ID should be a valid integer that is already in violations, otherwise, you can list the violations by calling WAF:violations. If the signature ID is not valid, then it will return "false", otherwise, it will return "true".
- WAF:raise_violation(table) on page 353 – Raises a violation immediately. This function will send a log by the input arguments. If the signature ID is already raised by the WAF then this command will override it.
- WAF:abandon_all() on page 356 – Abandons all of the results detected by the WAF module, including all of the violations, and resets the action to "pass".
- WAF:block(integer) on page 357 – Blocks the current session's client IP. Specify the period of the block in seconds as an integer (Range: 1-2147483647, default = 3600).
- WAF:unblock() on page 358 – Unblocks the client IP of the current session if it is already blocked.
- WAF:stage() on page 358 – The WAF:stage() method retrieves the current processing stage of the Web Application Firewall (WAF) module. This function helps identify which phase of WAF processing is currently executing, allowing for stage-specific logic and debugging.

WAF:enable()

Enables the current session's WAF scan function.

Syntax

WAF:enable();

Arguments

N/A

Events

Applicable in all WAF events:

- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
local s = WAF:status()
debug("test WAF_REQUEST_ATTACK_DETECTED, status %s\n", s)
WAF:enable()
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:disable()

Disables the current session's WAF scan function.

Syntax

WAF:disable();

Arguments

N/A

Events

Applicable in all WAF events:

- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
local s = WAF:status()
debug("test WAF_REQUEST_ATTACK_DETECTED, status %s\n", s)
WAF:disable()
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:status()

Returns a status string to specify the current status of WAF detection. The status may be "enable" or "disable".

Syntax

WAF:status();

Arguments

N/A

Events

Applicable in all WAF events:

- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
local s = WAF:status()
debug("test WAF_REQUEST_ATTACK_DETECTED, status %s\n", s)
WAF:disable()
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:action()

Returns the current session's WAF action.The return value is a string, which may include the following values:

- "pass"
- "deny"
- "block"
- "redirect"
- "captcha"

Syntax

WAF:action();

Arguments

N/A

Events

Applicable in all WAF events:

- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
local s = WAF:action()
debug("test WAF_REQUEST_ATTACK_DETECTED, action %s\n", s)
WAF:override_action("deny", 501);
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:override_action(action [, parameter])

Overrides the action that the WAF will take for the current transaction. This function is typically used within WAF event handlers to change the outcome, such as forcing a block, allowing a request, or challenging the client.

Syntax

WAF:override_action(action [, parameter]);

Arguments

The function requires an action string as the first argument. Certain actions require a second parameter.

Action	Second Parameter	Description
deny	code (Integer)	Terminates the request with the specified HTTP status code. Valid Codes: 200, 202, 204, 205, 400, 403, 404, 405, 406, 408, 410, 500, 501, 502, 503, 504. Default: 403 (if the code is invalid or not specified).
pass	None	Overrides any detected threat and allows the request to pass through.
captcha	None	Presents a CAPTCHA challenge to the client. The request is only allowed if the CAPTCHA is solved successfully.
block	period (Integer)	Blocks the client's IP address for the specified number of seconds.Range: 1 - 2147483647.Default: 3600 (if the period is invalid or not specified).
redirect	url (String)	Redirects the client to the specified URL. Note: The URL string must be valid and must be provided, otherwise the function will fail and return false.

Events

Applicable in all WAF events:

- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
local s = WAF:action()
debug("test WAF_REQUEST_ATTACK_DETECTED, action %s\n", s)
WAF:override_action("deny", 501);
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:violations()

Returns a table that includes all the violations detected by the current WAF stage as string values.

The table fields include the following:

Name	Description
severity	Includes the values "low", "medium", and "high".
information	The information that the WAF module defined when the specific attack was detected.
signature	An integer ID that is defined by the WAF module for every different attack.
action	The defined action is a violation, including the values "pass", "deny", "block", "redirect", or "captcha".
sub-category	The violation is related to a WAF sub-category field name. The string should be from the following list: • waf_web_attack_signature • waf_http_protocol_const • waf_heur_sqlxss_inject_detect • waf_url_protect,waf_bot_detection • waf_xml_check • waf_json_check • waf_web_scraping • waf_cookie_security • waf_csrf_protection • waf_html_input_validation • waf_brute_force,waf_data_leak_prevention • waf_credential_stuffing • waf_openapi_check • waf_api_gateway
owasp-top10	The violation is related to the OWASP TOP10 field name.

Syntax

WAF:violations();

Arguments

N/A

Events

Applicable in all WAF events:

- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
debug("test WAF_REQUEST_ATTACK_DETECTED\n")
local vl = WAF:violations();
for k, v in pairs(vl) do
debug("%d. Violation: signature %d, severity %s, information %s, action %s, sub-category %s,
owasp-top10 %s.\n", k, v["signature"], v["severity"], v["information"], v["action"], v["sub-
category"], v["owasp-top10"]);
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:abandon_violation(signature_id)

Removes a violation by the specified signature ID. The signature ID should be a valid integer that is already in violations, otherwise, you can list the violations by calling WAF:violations. If the signature ID is not valid, then it will return "false", otherwise, it will return "true".

This command can only be called in the ATTACK_DETECTED event.

Syntax

WAF:abandon_violation(signature_id);

Arguments

Name	Type	Description
signature_id	Integer	The signature ID of the violation to be removed. This should be a valid signature ID that exists in the current violations list.

Events

Applicable in all WAF events:

- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
debug("test WAF_REQUEST_ATTACK_DETECTED\n")

local vl = WAF:violations();
for k, v in pairs(vl) do
debug("%d. Violation: signature %d.\n", k, v["signature"]);
WAF:abandon_violation(v["signature"]);
end
v = {};
v["signature-id"] = 100010000;
v["severity"] = "high";
v["information"] = "waf raise violation test";
v["action"] = "deny";
v["sub-category"] = "waf_url_protect";
v["owasp-top10"] = "test-owasp10";
WAF:raise_violation(v);
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:raise_violation(table)

Raises a violation immediately. This function will send a log by the input arguments. If the signature ID is already raised by the WAF then this command will override it.

This function will prevent the WAF action from executing as specified. To override the WAF action, call WAF:override_action(string).

Syntax

WAF:raise_violation(table);

Arguments

Name	Description
severity	Overrides the severity string that includes the values "low", "medium", and "high". **Note**: If the value is not specified, then "low" will be used as the severity level for the violation.
information	The violation will show the information that the WAF module defined when the specific attack was detected. **Note:** If this is not specified, then it will show "N/A" as the violation's information.
signature	The attack signature string ID that WAF detected. Users can specify this if the signature ID already exists in the violation, which will override the related field of the violation by this function. **Note**: This argument must be specified.
action	The violation will show the defined action, including the values "pass", "deny", "block", "redirect", or "captcha". **Note**: If this is not specified, then the violation's action will take "pass" as default.
block-period	If the action is "block", then this argument must be specified. Otherwise, this will be defaulted to 3600. This argument should be an integer and range from 1-2147483647.
redirect-url	If the action is "redirect", then this argument must be specified. Otherwise, the "redirect" action will be ignored and will take a "deny" action instead.
deny-code	If the action is "deny", then this argument must be specified. The deny code should be an integer from the following: 200, 202, 204, 205, 400, 403, 404, 405, 406, 408, 410, 500, 501, 502, 503, 504. If the deny code is not specified or it is an invalid integer, then it will be defaulted to 403. The return value is a bool value; when the operation is successful, it will return true, otherwise, it will return false.

Name	Description
sub-category	This string specifies the violation's sub-category. The string should be from the following list: • waf_web_attack_signature • waf_http_protocol_const • waf_heur_sqlxss_inject_detect • waf_url_protect,waf_bot_detection • waf_xml_check • waf_json_check • waf_web_scraping • waf_cookie_security • waf_csrf_protection • waf_html_input_validation • waf_brute_force,waf_data_leak_prevention • waf_credential_stuffing • waf_openapi_check • waf_api_gateway **Note**: This argument is not required to be specified. But if this argument is not specified or if the string is not a valid sub-category, then it will default to "waf_web_attack_signature".
owasp-top10	The string will show the violation that is related to the OWASP TOP10 field name. **Note**: If this argument is not specified, then it will default to "unknown".

Events

Applicable in all WAF events:

- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
debug("test WAF_REQUEST_ATTACK_DETECTED\n")
local vl = WAF:violations();
for k, v in pairs(vl) do
debug("%d. Violation: signature %d.\n", k, v["signature"]);
WAF:abandon_violation(v["signature"]);
end
v = {};
v["signature-id"] = 100010000;
v["severity"] = "high";
v["information"] = "waf raise violation test";
v["action"] = "deny";
v["sub-category"] = "waf_url_protect";
v["owasp-top10"] = "test-owasp10";
WAF:raise_violation(v);
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:abandon_all()

Abandons all of the results detected by the WAF module, including all of the violations, and resets the action to "pass".

This command can only be called in the ATTACK_DETECTED event.

Syntax

WAF:abandon_all();

Arguments

N/A

Events

Applicable in all WAF events:

- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
debug("test WAF_REQUEST_ATTACK_DETECTED\n")
WAF:abandon_all()
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:block(integer)

Blocks the current session's client IP. Specify the period of the block in seconds as an integer (Range: 1-2147483647, default = 3600).

Syntax

WAF:block(integer);

Arguments

Name	Description
integer	An integer ranging from 1-2147483647.

Events

Applicable in all WAF events:

- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
debug("test WAF_REQUEST_ATTACK_DETECTED\n")
WAF:block(3600)
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:unblock()

Unblocks the client IP of the current session if it is already blocked.

Syntax

WAF:unblock();

Argument

N/A

Events

Applicable in all WAF events:

- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

Example

```
when WAF_REQUEST_BEFORE_SCAN {
local s = WAF:status()
debug("test WAF_REQUEST_BEFORE_SCAN, status %s\n", s)
WAF:unblock()
}
```

Supported Version

FortiADC version 6.2.x and later.

WAF:stage()

The WAF:stage() method retrieves the current processing stage of the Web Application Firewall (WAF) module. This function helps identify which phase of WAF processing is currently executing, allowing for stage-specific logic and debugging.

Syntax

WAF:stage();

Arguments

N/A

Events

Applicable in all WAF events:

- WAF_REQUEST_BEFORE_SCAN
- WAF_RESPONSE_BEFORE_SCAN
- WAF_REQUEST_ATTACK_DETECTED
- WAF_RESPONSE_ATTACK_DETECTED

```
Returns: A string representing the current WAF processing stage
Possible Values:
    "REQUEST_BEFORE_SCAN" - Before request scanning begins
    "REQUEST_ATTACK_DETECTED" - When an attack is detected in the request
    "RESPONSE_BEFORE_SCAN" - Before response scanning begins
    "RESPONSE_ATTACK_DETECTED" - When an attack is detected in the response.
```

Example

```
when WAF_REQUEST_ATTACK_DETECTED {
    local stage = WAF:stage()
    if stage == "REQUEST_ATTACK_DETECTED" then
        debug("Blocking malicious request at stage: %s\n", stage)
        -- Add custom blocking logic here
    end
}
```

Supported Version

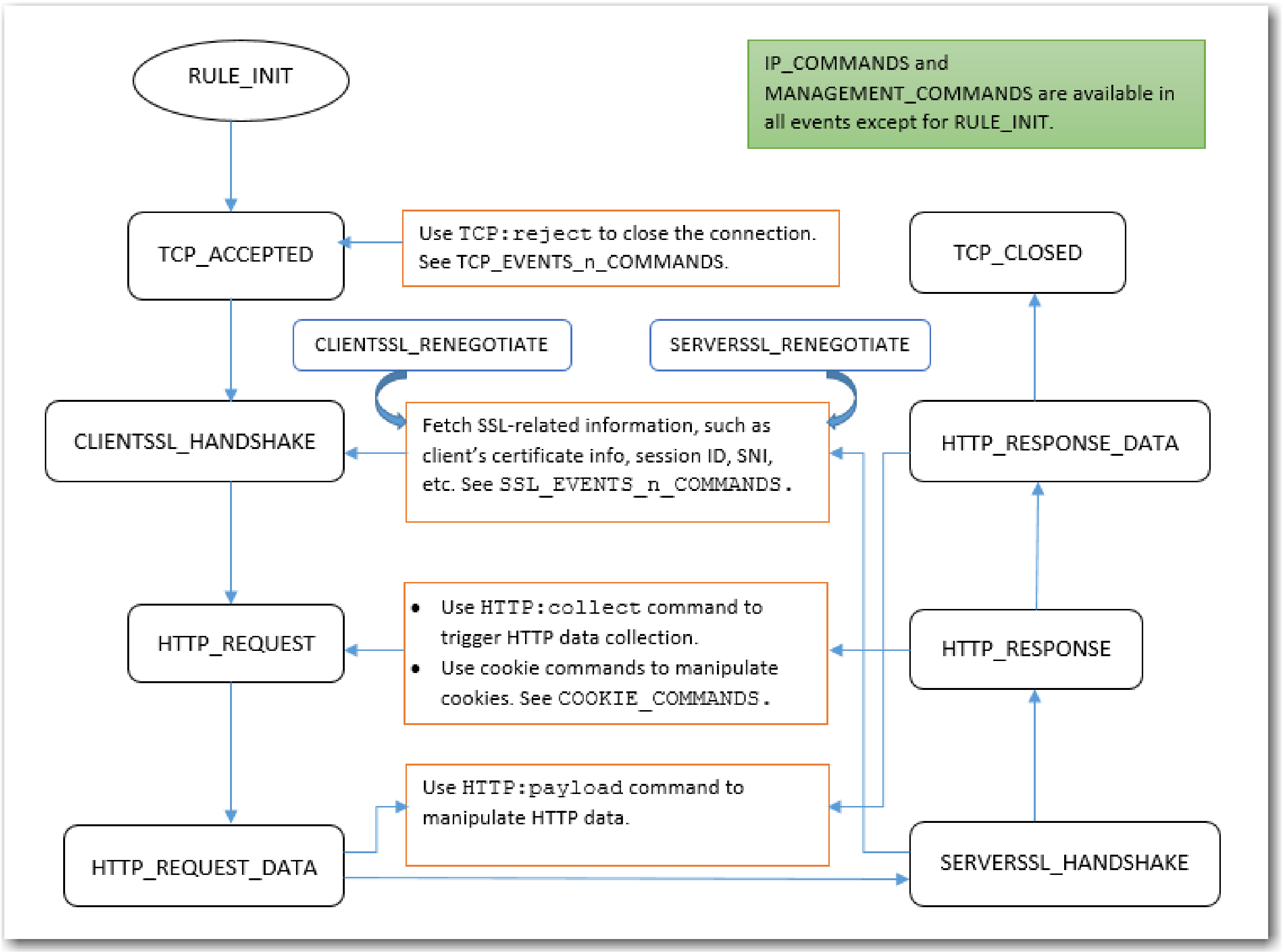
FortiADC version 6.2.x and later.

Predefined HTTP scripts

FortiADC provides system predefined scripts for HTTP Scripting.

Scripts and predefined commands on page 359 highlights the functions of these scripts and commands and shows how to use them.

Scripts and predefined commands


RULE_INIT
IP_COMMANDS and MANAGEMENT_COMMANDS are available in all events except for RULE_INIT.
TCP_ACCEPTED
Use TCP:reject to close the connection. See TCP_EVENTS_n_COMMANDS.
TCP_CLOSED
CLIENTSSL_RENEGOTIATE
SERVERSSL_RENEGOTIATE
CLIENTSSL_HANDSHAKE
Fetch SSL-related information, such as client's certificate info, session ID, SNI, etc. See SSL_EVENTS_n_COMMANDS.
HTTP_RESPONSE_DATA
HTTP_REQUEST
Use HTTP:collect command to trigger HTTP data collection.
Use cookie commands to manipulate cookies. See COOKIE_COMMANDS.
HTTP_RESPONSE
Use HTTP:payload command to manipulate HTTP data.
HTTP_REQUEST_DATA
SERVERSSL_HANDSHAKE

- UTILITY_FUNCTIONS_DEMO and CLASS_SEARCH_n_MATCH provide various utility commands.
- MULTIPLE_SCRIPT_CONTROL_DEMO_1 and MULTIPLE_SCRIPT_CONTROL_DEMO_2 show how to use multiple-script support.
- HTTP_DATA_FIND_REMOVE_REPLACE_DEMO and HTTP_DATA_FETCH_SET_DEMO show how to manipulate HTTP data.
- SPECIAL_CHARACTERS_HANDLING_DEMO shows how to handle certain special characters.
- INSERT_RANDOM_MESSAGE_ID_DEMO shows how to generate random message IDs.
- OPTIONAL_CLIENT_AUTHENTICATION shows how to perform optional client authentication based on a request URL.
- COMPARE_IP_ADDR_2_ADDR_GROUP_DEMO shows how to perform IP address match.
- USE_REQUEST_HEADERS_in_OTHER_EVENTS shows how to share information across events.
- Many more predefined scripts are provided for load balance content routing, HTTP redirection, and HTTP content rewriting.

The following table lists the FortiADC predefined scripts available for users to apply and customize.

Group	Predefined script	Usage
Authentication	AUTH_COOKIE_BAKE	Allows you to retrieve the baked cookie and edit the cookie content.

Group	Predefined script	Usage
	AUTH_EVENTS_n_COMMANDS	Lists the authentication event and commands.
	CUSTOMIZE_AUTH_KEY	Demonstrates how to customize the crypto key for authentication cookie.
	TWO_STEP_VERIFICATION	Demonstrates how to perform 2-Step Verification using FortiToken. One needs have authentication policy configured and selected in a virtual-server.
	TWO_STEP_VERIFICATION_2_NEW	Demonstrates how to perform 2-Step Verification using FortiToken for the second authentication group.
	TWO_STEP_VERIFICATION_2_SAME	Demonstrates how to perform 2-Step Verification for the second authentication group using the same token group.
	TWO_STEP_VERIFICATION_CHANGE_KEY	Demonstrates how to change the AES key and its size for stored token group.
Cookie	COOKIE_COMMANDS	Lists the two cookie commands and shows how to use them.
	COOKIE_COMMANDS_USAGE	Demonstrates the sub-function to handle the cookie attribute "SameSite" and others.
	COOKIE_CRYPTO_COMMANDS	Used to perform cookie encryption/decryption on behalf of the real server.
Feature	WAITING_ROOM	The sample Waiting Room script demonstrates how you can place visitors in a virtual queue instead of denying them service directly when the server side reaches its configured capacity limit during high-demand situations. In this virtual Waiting Room, visitors can see their position in line and when their turn arrives, they are redirected to the requested page. Configuration parameters include the waiting room name, total resource limit threshold (default is 1000), and the Resource URL applicable to the waiting room. You can also customize the message displayed to users when they are placed in the waiting room by editing the HTML page section of the script. Required data structures such as atomic counters and shared tables are already built into the script; however, you have the option to apply user-defined atomic counters and shared tables to customize the script.

Group	Predefined script	Usage
HTTP	GENERAL_REDIRECT_DEMO	Redirects requests to a URL with user-defined code and cookie. **Note**: Do **not** use this script "as is". Instead, copy and customize the code, URL, and cookie.
	HTTP_2_HTTPS_REDIRECTION	Redirects requests to the HTTPS site. **Note**: This script can be used directly without making any changes.
	HTTP_2_HTTPS_REDIRECTION_FULL_URL	Redirects requests to the specified HTTPS URL. **Note**: This script can be used directly without making any changes.
	HTTP_DATA_FETCH_SET_DEMO	Collects data in HTTP request body or HTTP response body. In `HTTP_REQUEST` or `HTTP_RESPONSE`, you could collect specified size data with "`size`" in `collect().In HTTP_DATA_REQUEST` or `HTTP_DATA_RESPONSE`. You could print the data use "`content`", calculate data length with "`size`", and rewrite the data with "`set`". **Note**: Do **not** use this script "as is". Instead, copy it and manipulate the collected data.
	HTTP_DATA_FIND_REMOVE_REPLACE_DEMO	Finds a specified string, removes a specified string, or replaces a specified string to new content in HTTP data. **Note**: Do **not** use this script "as is". Instead, copy it and manipulate the collected data.
	INSERT_RANDOM_MESSAGE_ID_DEMO	Inserts a 32-bit hex string into the HTTP header with a parameter "Message-ID". **Note**: This script can be used directly without making any changes.
	REDIRECTION_by_STATUS_CODE	Redirects requests based on the status code of server HTTP response (for example, a redirect to the mobile version of a site). **Note**: Do **not** use this script "as is". Instead, copy it and customize the condition in the server HTTP response status code and the URL values.
	REDIRECTION_by_USER_AGENT	Redirects requests based on User Agent (for example, a redirect to the mobile version of a site). **Note**: You should not use this script "as is". Instead, copy it and customize the User Agent and URL values.

Group	Predefined script	Usage
	REWRITE_HOST_n_PATH	Rewrites the host and path in the HTTP request, for example, if the site is reorganized. You should not use this script as is. Instead, copy it and customize the "old" and "new" hostnames and paths.

Group	Predefined script	Usage
	REWRITE_HTTP_2_HTTPS_in_LOCATION	Rewrites HTTP location to HTTPS, for example, `rewrite “Location:http://www.example.com” to “Location:https://www.example.com”.` **Note**: This script can be used directly without making any changes.
	REWRITE_HTTP_2_HTTPS_in_REFERER	Rewrites HTTP referer to HTTPS, for example, `rewrite “Referer: http://www.example.com” to “Referer: https://www.example.com”.` **Note**: This script can be used directly without making any changes.
	REWRITE_HTTPS_2_HTTP_in_LOCATION	Rewrites HTTPS location to HTTP, for example, `rewrite “Location:https://www.example.com” to “Location:http://www.example.com”.` **Note**: This script can be used directly without making any changes.
	REWRITE_HTTPS_2_HTTP_in_REFERER	Rewrites HTTPS referer to HTTP, for example, `rewrite “Referer: https://www.example.com” to “Referer: http://www.example.com”.` **Note**: This script can be used directly without making any changes.
	SPECIAL_CHARACTERS_HANDLING_DEMO	Shows how to use those "magic characters" which have special meanings when used in a certain pattern. The magic characters are ( ) . % + - * ? [ ] ^ $
	USE_REQUEST_HEADERS_in_OTHER_EVENTS	Stores a request header value in an event and uses it in other events. For example, you can store a URL in a request event, and use it in a response event. **Note**: Do **not** use this script "as is". Instead, copy it and customize the content you want to store, use `collect() in HTTP_REQUEST` to trigger `HTTP_DATA_REQUEST`,or use `collect() in HTTP_ RESPONSE` to trigger `HTTP_DATA_RESPONSE.`
IP	IP_COMMANDS	Used to get various types IP Address and port number between client and server side.

Group	Predefined script	Usage
Optimization	MULTIPLE_SCRIPT_CONTROL_DEMO_1	Uses `demo_1` and `demo_2` script to show how multiple scripts work. `Demo_1` with priority 12 has a higher priority. Note: You could enable or disable other events. Do NOT use this script "as is". Instead, copy it and customize the operation.
	MULTIPLE_SCRIPT_CONTROL_DEMO_2	Uses `demo_1` and `demo_2` script to show how multiple scripts work. `Demo_2` with priority 24 has a lower priority. **Note**: You can enable or disable other events. Do **not** use this script "as is". Instead, copy it and customize the operation.
	RAM_CACHING_COMMANDS	Lists the RAM caching event and commands.
	RAM_CACHING_DYNAMIC	Demonstrates how to use script to do dynamic RAM caching. **Note**: Dynamic caching is identified by a configured ID. Ensure the RAM caching configuration is selected in the HTTP or HTTPS profile.
	RAM_CACHING_GROUPING	Demonstrates how to create multiple variations based on client IP address. The sort of grouping applies to both regular caching and dynamic caching. **Note**: Ensure the RAM caching configuration is selected in HTTP or HTTPS profile.
Routing	CONTENT_ROUTING_by_URI	Routes to a pool member based on URI string matches. **Note**: You should not use this script as is. Instead, copy it and customize the URI string matches and pool member names.
	CONTENT_ROUTING_by_X_FORWARDED_FOR	Routes to a pool member based on IP address in the X-Forwarded-For header. **Note**: You should not use this script as is. Instead, copy it and customize the X-Fowarded-For header values and pool member names.
	PERSIST_COMMANDS	Demonstrates how to use persistence commands and event. The PERSISTENCE event is triggered when FortiADC receives the HTTP REQ and is ready to dispatch to the real server.

Group	Predefined script	Usage
		You can set the entry in PERSISTENCE, then look up it in POST_PERSIST. FortiADC will dispatch to the dedicated server according to your entry set in PERSISTENCE if this session has not been assigned to the real server before.
SSL	OPTIONAL_CLIENT_AUTHENTICATION	Performs optional client authentication. **Note**: Before using this script, you must have the following four parameters configured in the client-ssl-profile: • client-certificate-verify—Set to the verify you'd like to use to verify the client certificate. • client-certificate-verify-option—Set to optional • ssl-session-cache-flag—Disable. • use-tls-tickets—Disable.
	SSL_EVENTS_n_COMMANDS	Demonstrates how to fetch the SSL certificate information and some of the SSL connection parameters between server and client side.
TCP	SNAT_COMMANDS	Allows you to overwrite client source address to a specific IP for certain clients, also support IPv4toIPv6 or IPv6toIPv4 type. **Note**: Make sure the flag SOURCE ADDRESS is selected in the HTTP or HTTPS type of profile.
	SOCKOPT_COMMAND_USAGE	Allows user to customize the TCP_send buffer and TCP_receive buffer size.
	SOCKOPT_COMMANDS	Demonstrates how to the TCP:sockopt with usage examples.
	TCP_EVENTS_n_COMMANDS	Demonstrates how to reject a TCP connection from a client in TCP_ACCEPTED event.

Group	Predefined script	Usage
Utility	AES_DIGEST_SIGN_2F_COMMANDS	Demonstrates how to use AES to encryption/decryption data and some tools to generate the digest.
	ATOMIC_COUNTER_COMMANDS	Allows you to create and configure shared atomic counters that are accessible by multiple httpproxy processes within one VS. The stored data is located in shared memories. In the Waiting Room setup, the atomic counters track variables at running time, including the current resource count, the current position in line, and the current total number of users in the waiting queue.
	CLASS_SEARCH_n_MATCH	Demonstrates how to use the `class_match` and `class_search` utility function.
	COMPARE_IP_ADDR_2_ADDR_GROUP_DEMO	Compares an IP address to an address group to determine if the IP address is included in the specified IP group. For example ,192.168.1.2 is included in 192.168.1.0/24. **Note**: Do **not** use this script "as is". Instead, copy it and customize the IP address and the IP address group.
	GEOIP_UTILITY	Used to fetch the GEO information country and possible province name of an IP address.
	MANAGEMENT_COMMANDS	Allow you to disable/enable rest of the events from executing.
	SHARED_TABLE_COMMANDS	Allows you to create and configure shared hash tables that are accessible by multiple httpproxy processes within one VS. Both the table and stored data are located in shared memories. In the Waiting Room setup, the shared table is used to track current active resource occupiers such as active sessions.
	URL_UTILITY_COMMANDS	Demonstrates how to use those URL tools to encode/decode/parser/compare.
	UTILITY_FUNCTIONS_DEMO	Demonstrates how to use the basic string operations and random number/alphabet, time, MD5, SHA1, SHA2, BASE64, BASE32, table to string conversion, network to host conversion utility function
WAF	WAF_COMMANDS	Demonstrates how to use WAF related functions and events.

HTTP Scripting examples

This section provides use-case examples of HTTP Scripting application. It includes the following examples:

Select content routes based on URI string matches

The content routing feature has rules that match HTTP requests to content routes based on a Boolean AND combination of match conditions. If you want to select routes based on a Boolean OR, you can configure multiple rules. The content routing rules table is consulted from top to bottom until one matches.

In some cases, it might be simpler to get the results you want using a script. In the following example, each rule selects content routes based on OR match conditions.

Content routing example:

```
when RULE_INIT {
        debug("get header init 1\n")
}

when HTTP_REQUEST {
        uri = HTTP:uri_get()
        if uri:find("sports") or uri:find("news") or uri:find("government") then
                LB:routing("sp2")
                debug("uri %s matches sports|news|government\n", uri);
        elseif uri:find("finance") or uri:find("technology") or uri:find("shopping") then
                LB:routing("sp3")
                debug("uri %s matches finance|technology|shopping\n", uri);
        elseif uri:find("game") or uri:find("bbs") or uri:find("testing") then
                LB:routing("sp4")
                debug("uri %s matches game|bbs|testing\n", uri);
        elseif uri:find("billing") or uri:find("travel") or uri:find("weibo") then
                LB:routing("sp5")
                debug("uri %s matches billing|travel|weibo\n", uri);
        else
                debug("no matches for uri: %s \n", uri);
        end
}
```

To use a script for content routing:

1. Create the content route configuration objects. In the example above, sp2, sp3, sp4, and sp4 are the names of the content route configuration objects. You do not need to configure matching conditions for the content routes, however, because the script does the content matching.
2. Create a script that matches content to the content route configuration objects, as shown above. Create a configuration object for the script.
3. In the virtual server configuration:
   a. Enable content routing and select the content route configuration objects.
   b. Select the script.

Rewrite the HTTP request host header and path

You can use the content rewriting feature to rewrite the HTTP request Host header or the HTTP request URL. If you need more granular capabilities, you can use scripts. The following example rewrites the HTTP Host header and path.

Rewrite the HTTP Host header and path in a HTTP request:

```
when RULE_INIT {
        debug("rewrite the HTTP Host header and path in a HTTP request \n")
}

when HTTP_REQUEST {
       host = HTTP:header_get_value("Host")
       path = HTTP:path_get()
       if host:lower():find("myold.hostname.com") then
              debug("found myold.hostname.com in Host %s \n", host)
              HTTP:header_replace("Host", "mynew.hostname.com")
              HTTP:path_set("/other.html")
       end
}
```

Note: You might find it useful to use a combination of string manipulation functions. For example, this script uses lower() to convert the Host strings to lowercase in combination with find(), which searches for the Host header for a match: `host:lower():find("myold.hostname.com")`.

Rewrite the HTTP response Location header

You can use the content rewriting feature to rewrite the HTTP response Location header. If you are more comfortable using Lua string substitution, you can write a script to get the results you want. The following example rewrites the HTTP response Location header.

Rewrite the HTTP body in the response:

```
-- REWRITE_LOCATION_HTTP_TO_HTTPS.lua
-- Replace "http://" with "https://" in the Location response header.
when HTTP_RESPONSE {
    local loc = HTTP:header_get_value("Location")
    if loc and loc ~= "" then
        debug("Location header found: %s\n", loc)
        -- only rewrite if it starts with "http://"
        if string.find(loc, "^http://") then
            local newloc = string.gsub(loc, "^http://", "https://")
            debug("Location rewrite: %s -> %s\n", loc, newloc)
            HTTP:header_replace("Location", newloc)
        end
    else
        debug("Location header not found.\n")
    end
}
```

Redirect HTTP to HTTPS using Lua string substitution

You can use the content rewriting feature to redirect an HTTP request to an HTTPS URL that has the same host and request URL using a PCRE regular expression. If you are more comfortable using Lua string substitution, you can write a script to get the results you want. The following example redirects users to the HTTPS location.

Redirect HTTP to HTTPS:

```
when RULE_INIT {
        debug("http to https redirect\n")
}

when HTTP_REQUEST {
       host = HTTP:header_get_value("Host")
       path = HTTP:path_get()
       HTTP:redirect("https://%s%s",host,path);
}
```

Redirect mobile users to the mobile version of a website

The content rewriting feature does not support matching the User-Agent header. You can write a script that detects User-Agent headers that identify mobile device users and redirect them to the mobile version of a website.

Redirect mobile users to the mobile version of a website by parsing the User-Agent header:

```
when RULE_INIT {
        debug("detect User-Agent and go to mobile site\n")
}

when HTTP_REQUEST {
       path = HTTP:path_get()
       debug("path=%s\n",path)
       agent = HTTP:header_get_value("User-Agent")
       if agent:lower():find("iphone") or agent:lower():find("ipad")  then
              debug("found iphone or ipad in User-Agent %s \n", agent)
              HTTP:redirect("https://m.mymobilesite.com%s",path)
       end
}
```

Insert random message ID into a header

FortiADC offers the feature to insert messages and message IDs into HTTP request headers.

Insert a random message ID into the HTTP header:

```
when HTTP_REQUEST {
ID=HTTP:rand_id()-- a 32-long string of HEX symbols
HTTP:header_insert("Message-ID",ID)
}
```

General HTTP redirect

You can redirect both HTTP requests and HTTP responses to a given location.

Redirect an HTTP request:

```
when HTTP_REQUEST {
--can be used in both HTTP_REQUEST and HTTP_RESPONSE
--code and cookie are optional, code can be 301, 302, 303, 307, 308, if missed, 302 is used
t={}
t["code"] = 302;
t["url"] = "www.example.com"
t["cookie"] = "name=value; Expires=Wed, 09 Jun 2021 10:18:14 GMT"
HTTP:redirect_t(t);
}
```

Use request headers in other events

You can get stored request headers by using the session ID.

Use request headers in other events:

```
when RULE_INIT {
    --initialize the global so-called "environment" variable
    env={}
}
when HTTP_REQUEST {
    sess_id = HTTP:get_session_id()
    req={}
    --store whatever you want to req, take url as an example
    req["url"] = HTTP_uri_get()
    env[id] = req
}
when HTTP_RESPONSE {
    sess_id = HTTP:get_session_id()
    req = env[id]
    --now you can access the stored request headers
    debug("my stored request url is %s\n", req["url"]);
}
when HTTP_DATA_REQUEST {
    sess_id = HTTP:get_session_id()
    req = env[id]
    --now you can access the stored request headers
    debug("my stored request url is %s\n", req["url"]);
}
when HTTP_DATA_RESPONSE {
    sess_id = HTTP:get_session_id()
    req = env[id]
    --now you can access the stored request headers
    debug("my stored request url is %s\n", req["url"]);
}
```

Compare IP address to address group

You can compare IP addresses to an internal list of IP addresses. The script will return different results signifying whether the IP is in the list.

Compare an IP address to an address group:

```
when RULE_INIT {
--initialize the address group here
--for IPv4 address, mask can be a number between 0 to 32 or a dotted format
--support both IPv4 and IPv6, for IPv6, the mask is a number between 0 and 128
addr_group = "192.168.1.0/24"
addr_group = addr_group..",172.30.1.0/255.255.0.0"
addr_group = addr_group..",::ffff:172.40.1.0/120"
}
```

```
when HTTP_REQUEST {
client_ip = HTTP:client_addr()
matched = cmp_addr(client_ip, addr_group)
if matched then
debug("client ip found in address group\n");
else
debug("client ip not in address group\n");
end
}
```

Redirect HTTP to HTTPS

You can redirect an HTTP request from an HTTP location to an HTTPS location.

Redirect an HTTP request to HTTPS location:

```
when HTTP_REQUEST {
Host = HTTP:header_get_value("host")
Url = HTTP:uri_get()
HTTP:redirect("https://%s%s", Host, Url)
}
```

Rewrite HTTP to HTTPS in location

You can rewrite HTTP request headers to replace all HTTP addresses with HTTPS addresses in the redirect location.

Rewrite the HTTP request header to replace HTTP addresses with HTTPS in the redirect location:

```
when HTTP_RESPONSE {
loc = HTTP:header_get_value("Location")
if loc then
newloc = string.gsub(loc, "http", "https") --replace all http by https in the redirect location
HTTP:header_replace("Location", newloc);
end
}
```

Rewrite HTTP to HTTPS in referrer

You can rewrite HTTP request headers to replace all HTTP addresses with HTTPS addresses in the redirect referrer.

Rewrite the HTTP request header to replace HTTP addresses with HTTPS in the redirect referrer:

```
when HTTP_RESPONSE {
ref = HTTP:header_get_value("Referrer")
if ref then
newref = string.gsub(ref, "http", "https") --replace all http by https in the referrer header
HTTP:header_replace("Referer", newref);
end
}
```

Rewrite HTTPS to HTTP in location

You can rewrite HTTP request headers to replace all HTTPS addresses with HTTP addresses in the redirect location.

Rewrite the HTTP request header to replace HTTPS addresses with HTTP in the redirect location:

```
when HTTP_RESPONSE {
loc = HTTP:header_get_value("Location")
if loc then
newloc = string.gsub(loc, "https", "http") --replace all https by http in the redirect location
HTTP:header_replace("Location", newloc);
end
}
```

Rewrite HTTPS to HTTP in referer

You can rewrite HTTP request headers to replace all HTTPS addresses with HTTP addresses in the redirect referrer.

Rewrite the HTTP request header to replace HTTPS addresses with HTTP in the redirect referrer:

```
when HTTP_RESPONSE {
ref = HTTP:header_get_value("Referrer")
if ref then
newref = string.gsub(ref, "https", "http") --replace all https by http in the referrer header
HTTP:header_replace("Referrer", newref);
end
}
```

Fetch data from HTTP events

You can collect data from both HTTP request and HTTP response events. You can then manipulate this data.

Fetch data from HTTP events:

```
when HTTP_REQUEST {
--HTTP:collect command can be used in both HTTP_REQUEST and HTTP_RESPONSE events
--size is optional, otherwise, it will collect up to the full length or when 1.25M is reached
t={}
t["size"] = 100;
HTTP:collect(t)
}
```

```
when HTTP_DATA_REQUEST {
--check the size of the content
t={};
t["operation"]="size";
sz=HTTP:payload(t);
debug("content size: %s\n", sz);
--fetch the collected content
--offset and size are optional
t={};
t["operation"]="content";
t["offset"] = 0;
t["size"] = sz;
ct=HTTP:payload(t);
debug("content: %s\n", ct);
--do your own manipulation on the collected content
--replace the collected content by your new data
--offset and size are optional
t={};
t["operation"]="set";
t["offset"] = 0;
t["size"] = sz;
t["data"]="NEW DATA to SEND";
ret = HTTP:payload(t);
debug("set ret %s\n", ret);
}
```

Replace HTTP body data

You can find, remove, and replace data in the body of an HTTP request.

Replace the data in the HTTP request body:

```
when HTTP_REQUEST {
--HTTP:collect command can be used in both HTTP_REQUEST and HTTP_RESPONSE events
--size is optional, otherwise, it will collect up to the full length or when 1.25M is reached
t={}
t["size"] = 100;
HTTP:collect(t)
}
```

```
when HTTP_DATA_REQUEST {
--check the size of the content
t={};
t["operation"]="size";
sz=HTTP:payload(t);
debug("content size: %s\n", sz);
--find a string or a regular expression in the buffered data
--offset, size and scope are optional, if scope is missing, "all" is assumed
t={};
t["operation"]="find";
t["offset"] = 0;
t["size"] = sz;
t["scope"] = "all";-- or "first"
t["data"] = "your string or a regular expression to find";
if HTTP:payload(t) then
debug("found %d occurences\n", ret);
else
debug("not found\n");
end
--remove a string or a regular expression in the buffered data
--offset, size and scope are optional, if scope is missing, "all" is assumed
t={};
t["operation"]="remove";
t["offset"] = 0;
t["size"] = sz;
t["scope"] = "all";-- or "first"
t["data"] = "your string or a regular expression to find";
if HTTP:payload(t) then
debug("removed %d occurences\n", ret);
else
debug("not found\n");
end
--replace a string or a regular expression in the buffered data by a new string
--offset, size and scope are optional, if scope is missing, "all" is assumed
t={};
t["operation"]="replace";
t["offset"] = 0;
t["size"] = sz;
t["scope"] = "all";-- or "first"
t["data"] = "your string or a regular expression to find";
t["new_data"] = "your new data";
if HTTP:payload(t) then
debug("replaced %d occurences\n", ret);
else
debug("not found\n");
end
}
```

Set persistence table entry

You can set the entry to the persistence table and the real server will be assigned after lookup.

Set the entry to the persistence table:

```
when RULE_INIT {
    env={}
    PROXY:init_stick_tbl_timeout(1000)
}
```

```
when PERSISTENCE {
    debug("PERSIST \n");
    t={};
    t["operation"] = "get_valid_server";
    ret_tbl = HTTP: persist(t);
    if(ret_tbl) then
        for srv, state in pairs(ret_tbl) do
            debug("server %s status %s\n", srv, state);
        end
    end
    t={};
    t["operation"] = "save_tbl";
    t["hash_value"]= "hash_str";
    t["srv_name"]= "rsrv_70";
    ret = HTTP: persist(t)
    if ret then
        debug("save table success\n");
    else
        debug("save table failed\n");
    end
    t={};
    t["operation"] = "dump_tbl";
    t["index"] = 0;
    t["count"] = 500;
    ret_tbl = HTTP: persist(t)
    if(ret_tbl) then
        for k, cnt in pairs(ret_tbl) do
            debug(" hash %s srv_name %s\n", k, cnt)
        end
    end
    t={};
    t["hash_value"]= "hash_str";
    ret = HTTP:lookup_tbl(t);
    if ret then
        debug("LOOKUP success\n");
    else
        debug("LOOKUP fail\n");
    end
}
```

Assign server in PostPersist

You can get the current assigned server in PostPersist and assign the real server you desire by setting the table and lookup in PERSISTENCE.

Get the current assigned server in PostPersist to assign another real server:

```
when RULE_INIT {
    env={}
    PROXY:init_stick_tbl_timeout(1000)
}
```

```
when PERSISTENCE {
    debug("PERSIST \n");
    t={};
    t["hash_value"]= "hash_str";
    ret = HTTP:lookup_tbl(t);
    if ret then
        debug("LOOKUP success\n");
    else
        debug("LOOKUP fail\n");
    end
}
when POST_PERSIST {
    debug("POST PERSIST \n");
    t={};
    t["operation"] = "get_current_assigned_server"
    ret_tbl = HTTP: persist(t)
    if ret then
        debug("assign to %s\n", ret_tbl);
    else
        debug("get_current_assigned_server failed\n");
    end

    t={};
    t["operation"] = "save_tbl";
    t["hash_value"]= "hash_str";
    t["srv_name"]= "rsrv_70";
    ret = HTTP: persist(t)
    if ret then
        debug("save table success\n");
    else
        debug("save table failed\n");
    end
}
```

Run multiple scripts

You can run multiple scripts in FortiADC. When running multiple scripts, you may set a priority number for each script. FortiADC will run them in order from lowest priority to highest priority. The default priority is 500. If two scripts have the same priority number, they will be executed in the order in which they were added.

Run multiple scripts:

```
--script 1:
when HTTP_REQUEST priority 500 {
LB:routing("cr1")
}
--script 2:
when HTTP_RESPONSE priority 500 {
HTTP:close()
}
--script 3:
when HTTP_REQUEST priority 400 {
LB:routing("cr2")
}
--script 4:
when HTTP_RESPONSE priority 600 {
HTTP:close()
}
```

Set script priority in multi-script

While running multiple scripts, you can prioritize scripts. Add a priority number to each script when you create it, and FortiADC will run them in order from lowest priority to highest priority. The default priority is 500. If two scripts have the same priority number, they will be executed in the order in which they were added.

Set script priority:

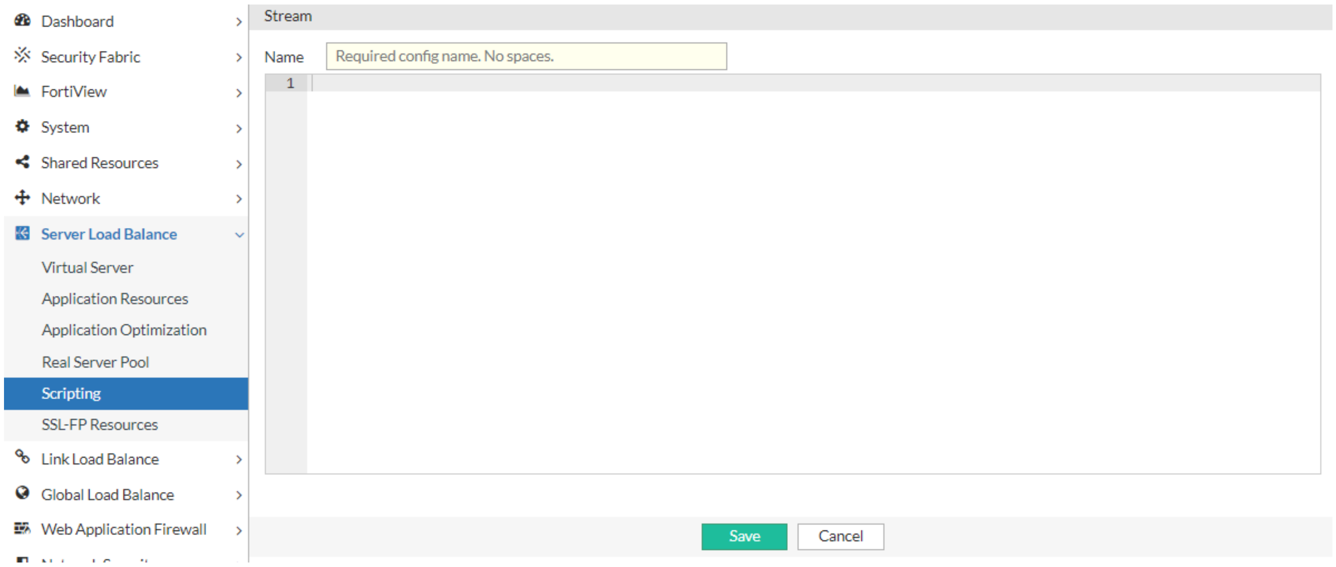
```
when RULE_INIT priority 14 {
--This is one of the script to demo the control of multiple scripts
--please change the priority of each event according to your need
debug("INIT in script 1\n");
}
when HTTP_REQUEST priority 12 {
debug("HTTP_REQUEST in script 1\n");
--add your own manipulation here
--you can disable rest of the HTTP_REQUEST events from executing by disabling this event
t={};
t["event"]="req"; -- can be "req", "res", "data_req", and "data_res"
t["operation"]="disable"; -- can be "enable", and "disable"
HTTP:set_event(t);
debug("disable rest of the HTTP_REQUEST events in script 1\n");
--you can also disable other events, say HTTP_RESPONSE, DATA events
--in the case of keep-alive, all events will be re-enabled automatically even though they are
disabled in previous TRANSACTION using the HTTP:set_event(t) command. To disable this automatic
re-enabling behavior, you can call HTTP:set_auto(t) as below
t={};
t["event"]="req"; -- can be "req", "res", "data_req", and "data_res"
t["operation"]="disable"; -- can be "enable", and "disable"
HTTP:set_auto(t);
debug("disable automatic re-enabling of the HTTP_REQUEST events in script 1\n");
--you can also disable automatic re-enabling for other events, say HTTP_RESPONSE, DATA events
}
```

or

```
when RULE_INIT priority 24 {
--This is one of the script to demo the control of multiple scripts
--please change the priority of each event according to your need
debug("INIT in script 2\n");
}
when HTTP_REQUEST priority 24 {
debug("HTTP_REQUEST in script 2\n");
--add your own manipulation here
--you can disable rest of the HTTP_REQUEST events from executing by disabling this event
t={};
t["event"]="req"; -- can be "req", "res", "data_req", and "data_res"
t["operation"]="disable"; -- can be "enable", and "disable"
HTTP:set_event(t);
debug("disable rest of the HTTP_REQUEST events in script 2\n");
--you can also disable other events, say HTTP_RESPONSE, DATA events
--in the case of keep-alive, all events will be re-enabled automatically even though they are
disabled in previous TRANSACTION using the HTTP:set_event(t) command. To disable this automatic
re-enabling behavior, you can call HTTP:set_auto(t) as below
t={};
t["event"]="req"; -- can be "req", "res", "data_req", and "data_res"
t["operation"]="disable"; -- can be "enable", and "disable"
HTTP:set_auto(t);
debug("disable automatic re-enabling of the HTTP_REQUEST events in script 2\n");
--you can also disable automatic re-enabling for other events, say HTTP_RESPONSE, DATA events
}
```

Stream Scripting

Stream Scripting in FortiADC allows you to perform actions that are not supported by the current built-in feature set for Layer 7 TCP and UDP virtual servers. Stream scripts enable you to use predefined script commands and variables to manipulate the TCP/UDP request and response. You can import Stream scripts from the FortiADC GUI. To get started, FortiADC provides system predefined scripts that can be cloned for customization.



This section covers the following:

- Stream Scripting configuration overview on page 382
- Stream Scripting events on page 389
- Predefined Stream Scripting commands on page 390
- Predefined Stream scripts on page 429

Stream Scripting configuration overview

You can use Stream scripts to perform actions that are not supported by the current built-in feature set for Layer 7 TCP and UDP virtual servers. Stream scripts enable you to use predefined script commands and variables to manipulate the TCP/UDP request and response.

Stream scripts are composed of several functional components that define the trigger events, commands, operators, and more. The following example demonstrates how Stream scripting is applied to return the client IP address of a connection for the frontend, it's source address for the backend, and it's destination address.

The Stream script:

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Stream script component breakdown:

Parameter	Example	Description
Events – for details, see Stream Scripting events on page 389.		
	STREAM_RESPONSE_DATA	When a response comes from a real server.
Commands – for details, see Predefined Stream Scripting commands on page 390.		
	IP:client_addr()	Returns the client IP address of a connection for the frontend, it's source address for the backend, and it's destination address.
	IP:local_addr()	For the frontend, this returns the IP address of the virtual

Parameter	Example	Description
		server that the client is connected to. For the backend, this returns the incoming interface IP address of the return packet.
	IP:remote_addr()	Returns the IP address of the host on the far end of the connection.
	IP:client_port()	Returns the local port number. In the frontend, the local port is the virtual server port. In the backend, the local port is the port of the gateway used to connect.
	IP:local_port()	Returns the local port number. In the frontend, the local port is the virtual server port. In the backend, the local port is the port of the gateway used to connect.
	IP:remote_port()	Returns the remote port number. In the frontend, the remote port is the client port. In the backend, the remote port is the real server port.
	IP:client_ip_ver()	Returns the current client IP version number of the connection, either 4 or 6.
	IP:server_addr()	Returns the IP address of the server in backend.
	IP:server_port()	Returns the server port number, which is the real server port.
	IP:server_ip_ver()	Returns the current server IP version number of the connection, either 4 or 6.
Operators – for details, see Operators on page 9. (Not applicable in this example).		
Strings – for details, see String library on page 15. (Not applicable in this example).		
Control structures – for details, see Control structures on page 8. (Not applicable in this example).		
Functions – for details, see Functions on page 13. (Not applicable in this example).		

Configuring Stream Scripting

From the FortiADC GUI, you can type or paste the script content into the configuration page. Alternatively, you can clone a system predefined script to customize. For details, see . From the HTTP Script page, you also have the option to import, export, and delete scripts.

Before you begin:

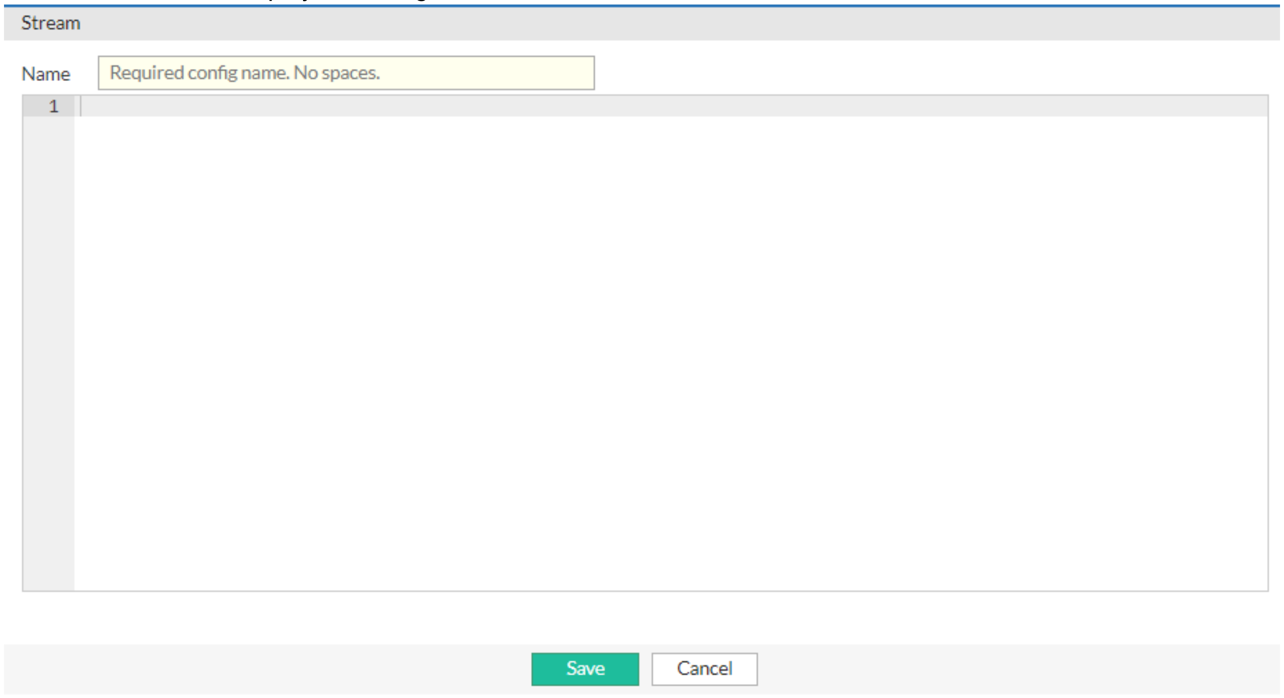
- You must have Read-Write permission for Server Load Balance settings.

After you have created a script configuration object, you can specify it in the virtual server configuration.

To create a Stream script configuration object:

1. Go to **Server Load Balance > Scripting**.
2. Click the **Stream** tab.

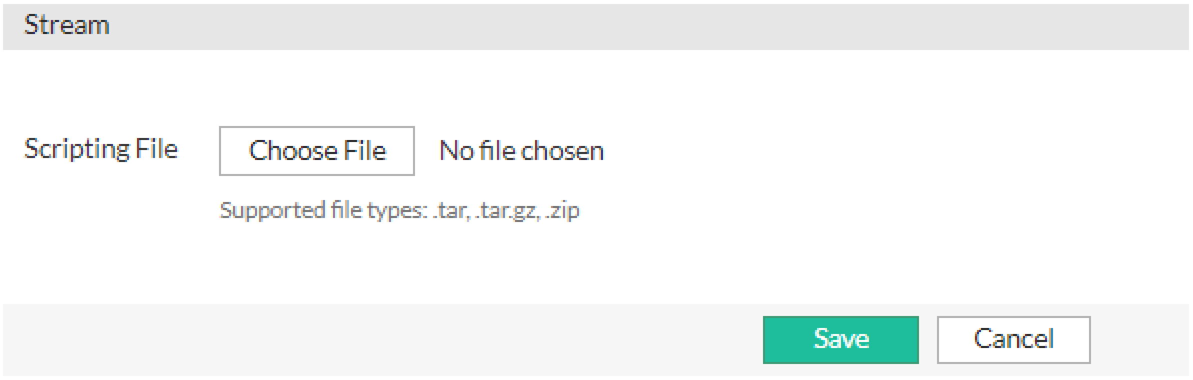
3. Click **Create New** to display the configuration editor.



4. Enter a unique name for the Stream script configuration. Valid characters are `A-Z`, `a-z`, `0-9`, `_`, and `-`. No spaces.
After you initially save the configuration, you cannot edit the name.
5. In the text box, type or paste your Stream script.
If you want to include this script as part of a multi-script configurations that allows you to execute multiple scripts in a certain order, ensure to set its priority. For more information, see Multi-script support for Stream Scripting on page 386.
6. Click **Save**.
Once the Stream script configuration is saved, you can specify it in the virtual server.

To import a Stream script:

1. Go to **Server Load Balance > Scripting**.
2. Click the **Stream** tab.
3. Click **Import** to display the file import options.

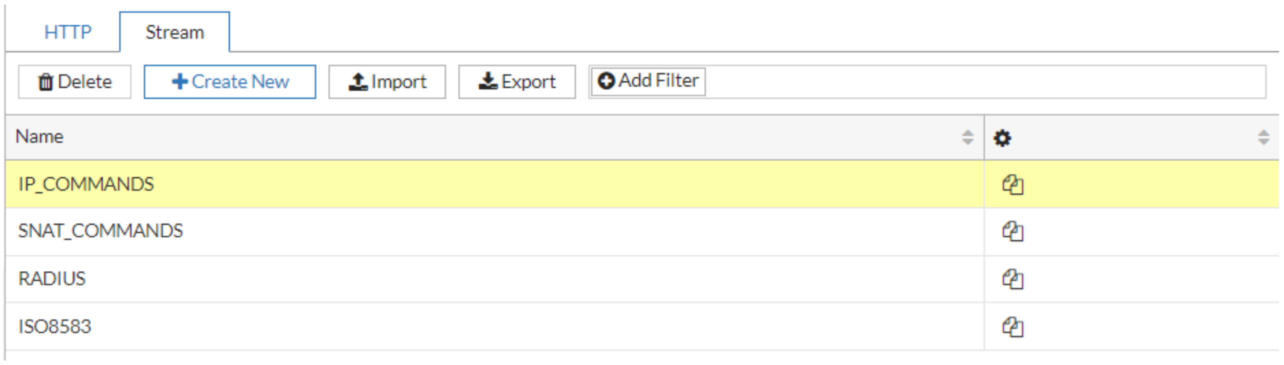


4. Click **Choose File** and browse for the script file. Supported file types are .tar, .tar.gz, and .zip.

5. Click **Save**.
   Once the file is successfully imported, it will be listed in the **Scripting > Stream** page.

To export a Stream script:

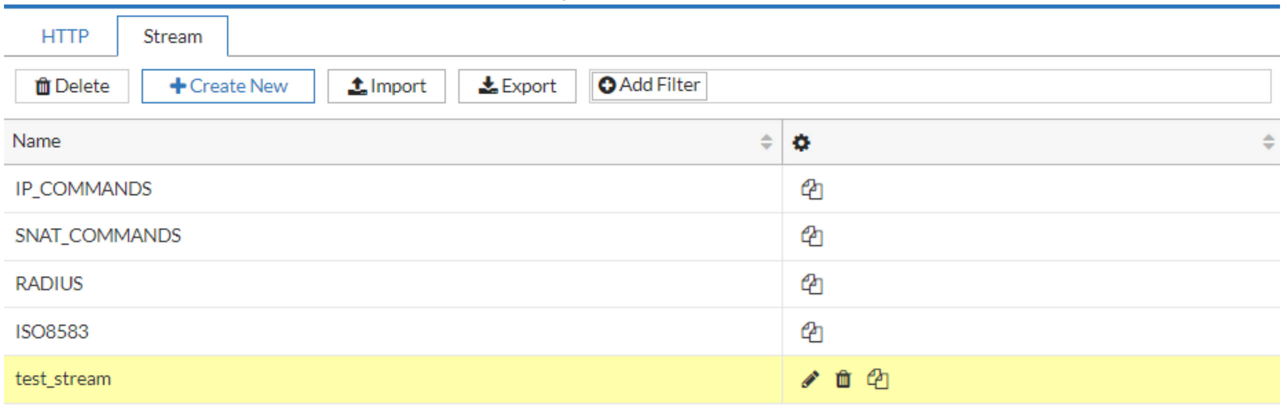
1. Go to **Server Load Balance > Scripting**.
2. Click the **Stream** tab.
3. From the **Stream** page, select a Stream script configuration.
   In the example below, the IP_COMMANDS script is selected.



4. Click **Export** initiate the file download.
   The selected script configuration will be exported as a .tar file.

To delete a Stream script:

1. Go to **Server Load Balance > Scripting**.
2. Click the **Stream** tab.
3. From the **Stream** page, select a user-defined Stream script configuration. System predefined scripts cannot be deleted.
   In the example below, the **test_stream** script configuration is selected.



4. Click **Delete** from the top navigation, or click 🗑 (delete icon) of the configuration.
   Multiple script configurations can be deleted using the **Delete** button on the top navigation.

Multi-script support for Stream Scripting

Linking multiple scripts to the same virtual server

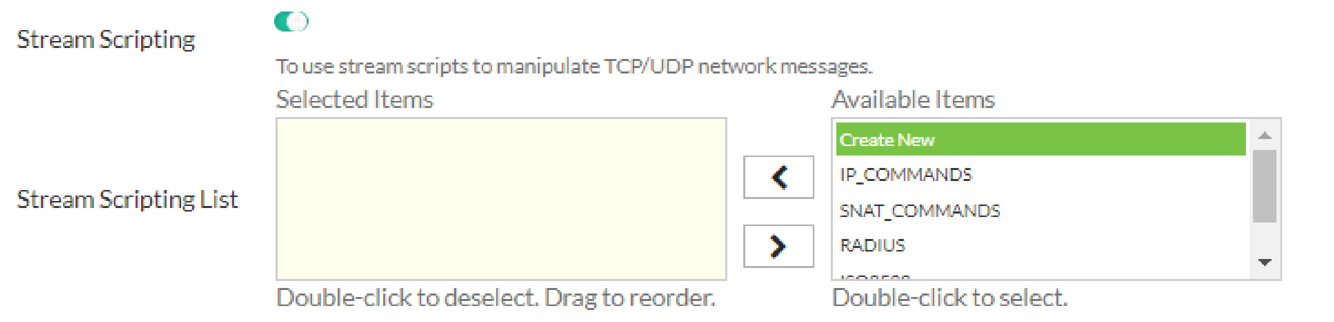
FortiADC supports the use of a single script file containing multiple scripts and applies them to a single virtual server in one execution. Different scripts can contain the same event. You can specify the priority for each event in each script file to control the sequence in which multiple scripts are executed or let the system to execute the individual scripts in the order they are presented in the multi-script file.

Currently, up to 16 individual scripts can be added to create a large multi-script file.

In practice, instead of creating a single large and complex script containing all necessary logic, it's often more advantageous to decompose it into smaller functional components represented by individual scripts. This approach offers several benefits. Firstly, executing multiple scripts concurrently is more efficient than running them sequentially. Additionally, breaking down a massive script into smaller units enhances flexibility, particularly when applying scripts to various virtual servers. Some servers may require only specific scripts, while others may utilize all available ones. With smaller, modular scripts, you have the flexibility to select and combine only the necessary components to construct a comprehensive multi-script file, each with its designated priority, and apply them collectively to a virtual server.

Apply multiple scripts on page 386 shows how to link multiple scripts to a single virtual server from the GUI.

Apply multiple scripts



Setting script priority

Priority in a multi-script is *optional*, but is highly recommended. When executing a big multiple-script file, care must be taken to avoid conflicting commands among the scripts. You can set the priority for each script using the script editor on FortiADC's GUI. Valid values range from 1 to 1,000, with 500 being the default. The smaller the value, the higher the priority. Below is an example script with a set priority:

```
when STREAM_CLIENT_INIT priority 100 {
    local header = TCP:peek(20)
}
```

To display the priority information in the GUI, you can define one and only one event in each script file, as shown below:

Script 1:

```
when STREAM_REQUEST_DATA priority 500 {
    local header = TCP:receive(20)
}
```

Script 2:

```
when STREAM_REQUEST_DATA priority 500 {
    local header = TCP:receive(20)
    local len = string.byte(string.sub(header, 3, 3))

    if (len < 2) then
        TCP:reject()
    end
}
```

Script 3:

```
when STREAM_REQUEST_DATA priority 400 {
    TCP:set_src_key(get_key())
}
```

Script 4:

```
when STREAM_RESPONSE_DATA priority 600 {
    TCP:set_dst_key(get_key())
}
```

Individual script files are loaded separately into the Lua stack. A numeric value (starting from 1) is appended to each event (e.g., for STREAM_REQUEST_DATA event, there are functions `STREAM_REQUEST_DATA1`, `STREAM_REQUEST_DATA2,` and so on so forth).

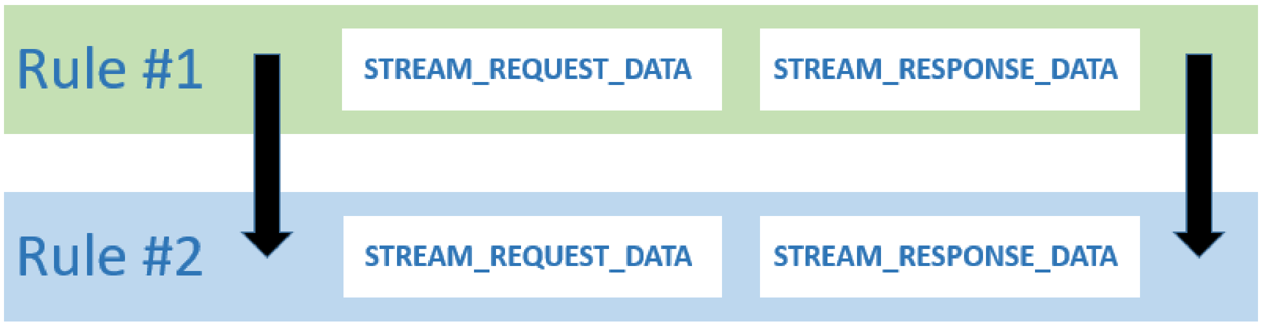
To support multiple scripts, FortiADC:

- Supports multiple calls of redirect/routing/close function, making them re-entrant so that the final one prevails. For that purpose, the system checks the behavior of multiple calls across `redirect(), close(), and routing()`. If `redirect()` comes first, followed by `close(),` then `close()` prevails. If `close()` comes first, followed by `redirect()`, then `redirect()` prevails. If you want to `close()`, you must disable the event after `close()`.
- Allows enabling or disabling events. There are times when you may want to disable the processing of the remaining scripts while a multi-script file is being executed, or want to disable processing the response completely. The mechanism serves that purpose.

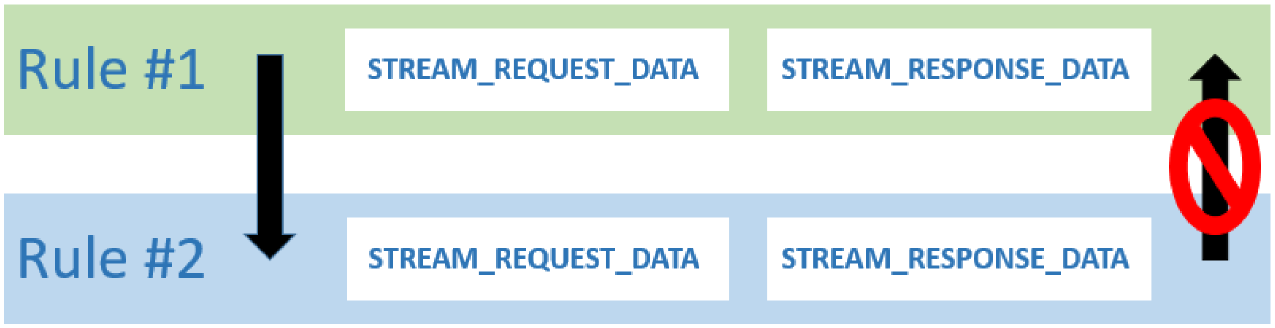
Compiling principles

- All individual scripts should be pre-compiled when they are linked to a virtual server, where they can be combined into one big multi-script.
- For the same event, combine the commands in different scripts according to their priorities and orders.
- For commands of different priorities, FortiADC processes the high-priority commands first, and then the low-priority ones; for commands of the same priority, it processes them in the order they appear in the combined script.
- And if you are using multiple scripts with overlapping events for bidirectional traffic, you must ensure that the response traffic traverses the overlapping events in the expected order. By default, the scripts applied to the same virtual server will run in the order in which they are applied, regardless of the direction of traffic flow.

- For a specified event, you must make sure to avoid the conflict commands in different scripts. For example, if you have multiple scripts applied to the same virtual server and the scripts contain both request and response logic, the default execution order is like this:



but NOT like this:



As shown above, FortiADC cannot control the order in which events in the scripts are executed. The only way to enforce the execution order for response traffic is to use the event priority command, as we have discussed above. When setting the priorities, pay special attention to both request and response flows.

Special notes

When using the multi-script feature, keep the following in mind:

- The multi-script feature is supported on all FortiADC hardware platforms.
- Currently, the feature can be applied to Layer 7 virtual servers on TCP or UDP protocols only.
- Scripts are VDOM-specific, and cannot be shared among different VDOMs.
- Session tables set up using scripts must be synced through high-availability (HA) configuration.
- Each multi-script configuration can contain up to 256 individual scripts, each being no more than 32 kilobytes.

Stream Scripting events

Event Name	Description	Available Protocol
RULE_INIT	When initializing the script.	TCP/UDP
STREAM_CLIENT_INIT	When a connection from a client is initialized.	TCP/UDP
STREAM_REQUEST_DATA	When a request comes from a client.	TCP/UDP
STREAM_RESPONSE_DATA	When a response comes from a real server.	TCP/UDP

Predefined Stream Scripting commands

IP commands

- IP:client_addr() on page 392 – Returns the client IP address of a connection. For the frontend, this refers to the source address; for the backend, this refers to the destination address.
- IP:server_addr() on page 394 – Returns the IP address of the server in the backend.
- IP:local_addr() on page 395 – For the frontend, this returns the IP address of the virtual server that the client is connected to. For the backend, this returns the incoming interface IP address of the return packet.
- IP:remote_addr() on page 397 – Returns the IP address of the host on the far end of the connection.
- IP:client_port() on page 399 – Returns the local port number. In the frontend, the local port is the virtual server port. In the backend, the local port is the port of the gateway used to connect.
- IP:server_port() on page 401 – Returns the server port number, which is the real server port.
- IP:local_port() on page 403 – Returns the local port number. In the frontend, the local port is the virtual server port. In the backend, the local port is the port of the gateway used to connect.
- IP:remote_port() on page 405 – Returns the remote port number. In the frontend, the remote port is the client port. In the backend, the remote port is the real server port.
- IP:client_ip_ver() on page 407 – Returns the current client IP version number of the connection, either 4 or 6.
- IP:server_ip_ver() on page 409 – Returns the current server IP version number of the connection, either 4 or 6.

IP:client_addr()

Returns the client IP address of a connection. For the frontend, this refers to the source address; for the backend, this refers to the destination address.

Syntax

cip=IP:client_addr()

Arguments

N/A

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:server_addr()

Returns the IP address of the server in the backend.

Syntax

sip=IP:server_addr()

Arguments

N/A

Events

Applicable in STREAM_RESPONSE_DATA.

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:local_addr()

For the frontend, this returns the IP address of the virtual server that the client is connected to. For the backend, this returns the incoming interface IP address of the return packet.

Syntax

sip=IP:local_addr()

Arguments

N/A

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:remote_addr()

Returns the IP address of the host on the far end of the connection.

Syntax

sip=IP:remote_addr()

Arguments

N/A

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:client_port()

Returns the local port number. In the frontend, the local port is the virtual server port. In the backend, the local port is the port of the gateway used to connect.

Syntax

cp=IP:client_port()

Arguments

N/A

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:server_port()

Returns the server port number, which is the real server port.

Syntax

sp=IP:server_port()

Arguments

N/A

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:local_port()

Returns the local port number. In the frontend, the local port is the virtual server port. In the backend, the local port is the port of the gateway used to connect.

Syntax

sp=IP:local_port()

Arguments

N/A

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:remote_port()

Returns the remote port number. In the frontend, the remote port is the client port. In the backend, the remote port is the real server port.

Syntax

rp=IP:remote_port()

Arguments

N/A

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:client_ip_ver()

Returns the current client IP version number of the connection, either 4 or 6.

Syntax

cv=IP:client_ip_ver ()

Arguments

N/A

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

IP:server_ip_ver()

Returns the current server IP version number of the connection, either 4 or 6.

Syntax

cv=IP:server_ip_ver ()

Arguments

N/A

Events

Applicable in STREAM_RESPONSE_DATA.

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

TCP commands

- TCP:peek() on page 411 – Allows you to get data from a TCP connection before connecting to a real server.
- TCP:receive() on page 412 – Allows you to get data from a TCP connection.
- TCP:reject() on page 413 – Allows you to reject a TCP connection from a client.
- TCP:set_snat_ip(str) on page 414 – Allows you to set the backend TCP connection's source address and port.
- TCP:set_src_key(data) on page 415 – Allows you to set a source key. FortiADC uses this key to identify the TCP connection in scenarios where there are multiple clients to one real server.
- TCP:set_dst_key(data) on page 416 – Allows you to set a destination key. FortiADC uses this key to identify the TCP connection in scenarios where there are multiple clients to one real server.
- TCP:isolate_client_close(flag) on page 417 – Allows you to isolate the real server connection in the event the client connection closes. If the flag is 1, then FortiADC will not close the real server the client closes.
- TCP:isolate_server_error(flag) on page 418 – Allows you to isolate the client connection in the event of a real server error. If the flag is 1, then FortiADC will not close the client when the real server sends a RESET to FortiADC.

TCP:peek()

Allows you to get data from a TCP connection before connecting to a real server.

Syntax

TCP:peek(size)

Arguments

Name	Description
size	The data size that you want to get from a TCP connection.

Events

Applicable in STREAM_CLIENT_INIT.

Example

```
when STREAM_CLIENT_INIT {
    local header = TCP:peek(20)
}
```

Supported Version

FortiADC version 6.1.1 and later.

TCP:receive()

Allows you to get data from a TCP connection.

Syntax

TCP:receive(size)

Arguments

Name	Description
size	The data size that you want to get from a TCP connection.

Events

Applicable in the following events:

- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_REQUEST_DATA {
    local header = TCP:receive(20)
}
```

Supported Version

FortiADC version 6.1.1 and later.

TCP:reject()

Allows you to reject a TCP connection from a client.

Syntax

TCP:reject()

Arguments

N/A

Events

Applicable in STREAM_REQUEST_DATA.

Example

```
when STREAM_REQUEST_DATA {
    local header = TCP:receive(20)
    local len = string.byte(string.sub(header, 3, 3))

    if (len < 2) then
        TCP:reject()
    end
}
```

Supported Version

FortiADC version 6.1.1 and later.

TCP:set_snat_ip(str)

Allows you to set the backend TCP connection's source address and port.

Syntax

TCP:set_snat_ip(str)

Arguments

Name	Description
str	A string which specifies the IP address and port.

Events

Applicable in STREAM_CLIENT_INIT.

Example

```
when STREAM_CLIENT_INIT {
    TCP:set_snat_ip(IP:remote_addr())
}
```

Supported Version

FortiADC version 6.1.1 and later.

TCP:set_src_key(data)

Allows you to set a source key. FortiADC uses this key to identify the TCP connection in scenarios where there are multiple clients to one real server.

Syntax

TCP:set_src_key(data)

Arguments

Name	Description
data	A string which can identify a TCP connection.

Events

Applicable in STREAM_REQUEST_DATA.

Example

```
when STREAM_REQUEST_DATA {
    TCP:set_src_key(get_key())
}
```

Supported Version

FortiADC version 6.1.1 and later.

TCP:set_dst_key(data)

Allows you to set a destination key. FortiADC uses this key to identify the TCP connection in scenarios where there are multiple clients to one real server.

Syntax

TCP:set_dst_key(data)

Arguments

Name	Description
data	A string which can identify a TCP connection.

Events

Applicable in STREAM_RESPONSE_DATA.

Example

```
when STREAM_RESPONSE_DATA {
    TCP:set_dst_key(get_key())
}
```

Supported Version

FortiADC version 6.1.1 and later.

TCP:isolate_client_close(flag)

Allows you to isolate the real server connection in the event the client connection closes. If the flag is 1, then FortiADC will not close the real server the client closes.

Syntax

TCP:isolate_client_close(flag)

Arguments

Name	Description
flag	• 1: Isolate client close. • 0: Does not isolate client close (default).

Events

Applicable in RULE_INIT.

Example

```
when RULE_INIT {
    TCP:isolate_client_close(1)
}
```

Supported Version

FortiADC version 6.1.1 and later.

TCP:isolate_server_error(flag)

Allows you to isolate the client connection in the event of a real server error. If the flag is 1, then FortiADC will not close the client when the real server sends a RESET to FortiADC.

Syntax

TCP:isolate_server_error(flag)

Arguments

Name	Description
flag	• 1: Isolate server error. • 0: Does not isolate server error (default).

Events

Applicable in RULE_INIT.

Example

```
when RULE_INIT {
    TCP:isolate_server_error(1)
}
```

Supported Version

FortiADC version 6.1.1 and later.

UDP commands

- UDP:receive() on page 420 – Allows you to get data from a UDP datagram.
- UDP:reject() on page 421 – Allows you to reject a UDP datagram from a client.
- UDP:set_snat_ip(str) on page 422 – Allow you to set the backend UDP datagram's source address and port.

UDP:receive()

Allows you to get data from a UDP datagram.

Syntax

UDP:receive(size)

Arguments

Name	Description
size	The data size that you want to get from a UDP datagram.

Events

Applicable in the following events:

- STREAM_REQUEST_DATA
- STREAM_RESPONSE_DATA

Example

```
when STREAM_REQUEST_DATA {
    local header = UDP:receive(20)
}
```

Supported Version

FortiADC version 6.1.1 and later.

UDP:reject()

Allows you to reject a UDP datagram from a client.

Syntax

UDP:reject()

Arguments

N/A

Events

Applicable in STREAM_REQUEST_DATA.

Example

```
when STREAM_REQUEST_DATA {
    local header = UDP:receive(20)
    local len = string.byte(string.sub(header, 3, 3))

    if (len < 2) then
        UDP:reject()
    end
}
```

Supported Version

FortiADC version 6.1.1 and later.

UDP:set_snat_ip(str)

Allow you to set the backend UDP datagram's source address and port.

Syntax

UDP:set_snat_ip(str)

Arguments

Name	Description
str	A string which specifies the IP address and port.

Events

Applicable in STREAM_CLIENT_INIT.

Example

```
when STREAM_CLIENT_INIT {
    UDP:set_snat_ip(IP:remote_addr())
}
```

Supported Version

FortiADC version 6.1.1 and later.

LB commands

- LB:upstream(real_server) on page 424 – Allows you to select a real server.
- LB:set_peer(ip, port) on page 425 – Allows you to select a real server with a specific IP and port.

LB:upstream(real_server)

Allows you to select a real server.

Syntax

LB:upstream(real_server)

Arguments

Name	Description
real_server	A real server name. **Note**: Only a real server from the regular pool can be used here; it cannot use a real server from the schedule pool. If a real server is not specified, then FortiADC will select a real server by using the specified load-balancing method.

Events

Applicable in the following events:

- STREAM_CLIENT_INIT
- STREAM_REQUEST_DATA

Example

```
when STREAM_REQUEST_DATA {
    LB:upstream("real-server-01")
}
```

Supported Version

FortiADC version 6.1.1 and later.

LB:set_peer(ip, port)

Allows you to select a real server with a specific IP and port.

Syntax

LB:set_peer(ip, port)

Arguments

Name	Description
ip	A real server IP address. **Note**: Only the IP address in a real server from the regular pool can be used here; it cannot use a real server from the schedule pool. If a real server is not specified, then FortiADC will select a real server by using the specified load-balancing method.
port	A real server port. **Note**: Only the port in a real server from the regular pool can be used here; it cannot use a real server from the schedule pool. If a real server is not specified, then FortiADC will select a real server by using the specified load-balancing.

Events

Applicable in STREAM_REQUEST_DATA.

Example

```
when STREAM_REQUEST_DATA {
    LB:set_peer("10.0.0.1","443")
}
```

Supported Version

FortiADC versions 7.2.7, 7.4.5, 7.6.1 and later.

Global commands

- debug(str) on page 427 – Prints the debug information when the virtual server is using stream scripting.
- log(str) on page 428 – Prints the scripting running information in log format. When using this command, you should enable scripting log.

debug(str)

Prints the debug information when the virtual server is using stream scripting.

Syntax

debug(str)

Arguments

Name	Description
str	A string which will be printed.

Events

Applicable in **all** events.

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

log(str)

Prints the scripting running information in log format. When using this command, you should enable scripting log.

Syntax

log(str)

Arguments

Name	Description
str	A string which will be logged.

Events

Applicable in **all** events.

Example

```
when STREAM_RESPONSE_DATA {
    local cip=IP:client_addr()
    local lip=IP:local_addr()
    local rip=IP:remote_addr()
    local cp=IP:client_port()
    local lp=IP:local_port()
    local rp=IP:remote_port()
    local cipv=IP:client_ip_ver()

    local sip=IP:server_addr()
    local sp=IP:server_port()
    local sipv=IP:server_ip_ver()

    debug("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s\n", rip, rp, cip, cp,
lip, lp, cipv)
    debug("resp: server %s:%s, sip version %s\n", sip, sp, sipv)
    log("resp: remote %s:%s, client %s:%s, local %s:%s, cip version %s", rip, rp, cip, cp, lip,
lp, cipv)
    log("resp: server %s:%s, sip version %s", sip, sp, sipv)
}
```

Supported Version

FortiADC version 6.1.1 and later.

Predefined Stream scripts

FortiADC provides system predefined scripts for Stream Scripting.

The following table lists the FortiADC predefined scripts available for users to apply and customize.

Predefined script	Usage
IP_COMMANDS	Used to get various types of IP addresses and port numbers between the client and server side.
SNAT_COMMANDS	Allows you to overwrite client source address to a specific IP for certain clients.
ISO8583	Implements ISO8583 load balancing based on message content.
RADIUS	Implements RADIUS load balancing based on message content.

www.fortinet.com