Script Reference Guide

FortiWeb 7.2.0

FORTINET DOCUMENT LIBRARY
HTTPs://docs.fortinet.com

FORTINET VIDEO GUIDE
HTTPs://video.fortinet.com

FORTINET BLOG
HTTPs://blog.fortinet.com

CUSTOMER SERVICE & SUPPORT
HTTPs://support.fortinet.com

FORTINET COOKBOOK
HTTPs://cookbook.fortinet.com

FORTINET TRAINING & CERTIFICATION PROGRAM
HTTPs://www.fortinet.com/support-and-training/training.html

NSE INSTITUTE
HTTPs://training.fortinet.com

FORTIGUARD CENTER
HTTPs://fortiguard.com/

END USER LICENSE AGREEMENT
HTTPs://www.fortinet.com/doc/legal/EULA.pdf

FEEDBACK
Email: techdocs@fortinet.com

August 22, 2022
FortiWeb 7.2.0 Script Reference Guide
1st Edition

TABLE OF CONTENTS

Change Log

Date	Change Description
2022-08-21	Initial release.

Introduction

FortiWeb supports Lua scripts to perform actions that are not currently supported by the built-in feature set. You can use Lua scripts to write simple, network aware pieces of code that will influence network traffic in a variety of ways. By using the scripts, you can customize FortiWeb's features by granularly controlling the traffic flow or even the contents of given sessions or packets.

In FortiWeb, the scripting language only supports HTTP and HTTPS policy.

Configuration overview

You can type or paste the script content into the configuration page.

Before you begin:

- Create a script.
- You must have Read-Write permission for **Server Policy** settings.

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

To configure a script:

1. Go to **Application Delivery > Scripting**.
2. Click **Create New** to display the configuration editor.
3. Complete the configuration as shown.

Settings	Guidelines
Name	Enter a unique name. No spaces or special characters. After you initially save the configuration, you cannot edit the name.
Input	Type or paste the script.

4. Click **OK** to Save the configuration.
5. You can also click **Import** to import a script file. It should be a ".txt" file.
6. When creating a server policy, in the **Scripting** section, enable **Scripting**, then select the scripts you want to run for this server policy.

Script Events

There are predefined scripts which specify the following events. When the events occur, it will trigger the system to take the actions defined in the script.

Type	Event	Description
RULE	RULE_INIT	When the server policy enables or reloads.
	RULE_EXIT	When the server policy disables or reloads.
HTTP	HTTP_REQUEST	When the server policy has received the complete HTTP request header.
	HTTP_RESPONSE	When the server policy has received the complete HTTP response header.
	HTTP_DATA_REQUEST	When an `HTTP:collect` command finishes processing, after collecting the requested amount of data.
	HTTP_DATA_RESPONSE	When an `HTTP:collect` command finishes processing on the server side of a connection.

Type	Event	Description
TCP	CLIENT_ACCEPTED	When the server policy has accepted a client connection.
	CLIENT_CLOSED	When the server policy has closed a client connection.
	SERVER_CONNECTED	When the server policy has connected to a server.
	SERVER_CLOSED	When the server policy has closed a server connection.

Event priority

FortiWeb supports multiple scripts in one server policy. When a server policy with scripts is enabled, the system will load scripts one by one. If there are multiple same events defined in the scripts, the event running order is same as the loading order.

If you want to run a certain event first regardless of the script order, you can define its priority to prioritize its sequence. The default priority of events is 500. Lower value has higher priority.

For example:

```
when HTTP_REQUEST priority 499 {
...
}
```

Lua package compatibility

FortiWeb uses the lua version 5.4.

Package name	Compatible details
global	Supported, but: • Disable dofile() • Disable loadfile() • Modify print() to FortiWeb version, printing to debug log with level 1. (`diag debug proxyd scripting-user <1-7>`)
package	Disabled
coroutine	Disabled
table	Supported
io	Disabled
os	Disabled
string	Supported
math	Supported
utf8	Supported

Predefined packages and classes

Global

debug(fmt, ..)

The function is the same as

```
print(string.format(fmt, ..))
```

The string will be printed to debug log with level 1. For example:

```
debug("This HTTP Request method is %s.\n", HTTP:method())
```

_id

This is the id of the proxyd worker running the lua stack.

_name

This is the name of the policy running the lua stack.

Core

core.debug(level, fmt, ..)

It is similar to `debug()` but it can set the debug log level.

core.print(level, …)

It is similar to `print()` but it can set the debug log level.

Policy

This package is used for fetching the policy configurations.

policy.name()

Return the string of the policy name.

policy.http_ports()

Return a lua array with all HTTP ports. Port value is integer.

```
{ 80, 8080 }
```

policy.https_ports()

Return a lua array with all HTTPS port. Port value is integer.

```
{ 443, 8443 }
```

policy.crs()

Return lua array with all content routing names.

```
{ "cr1", "cr2", "cr3" }
```

policy.servers() / policy.servers("cr-name")

Return lua array with all servers. If the policy has content routing, the caller should pass the "cr-name" argument to fetch the servers of the specific content routing.

```
{
   { ["type"] = "ip", ["ip"] = "172.30.154.2", ["port"] = 80 },
   { ["type"] = "ip", ["ip"] = "172.30.154.3", ["port"] = 80 },
   ...
}
```

IP

This package contains IP related functions.

ip.addr(“ip-string”)

Generate an IP address class with an IP string.

ip.eq(ip_class_1, “ip-string”) / ip.eq(ip_class_1, ip_class_2)

Compare two IP addresses. The first one must be IP address class and the second one can be IP address class or IP string.

ip.reputation(“ip-string”) / ip.reputation(ip_class)

Check the reputation of a specific IP. Return Lua array with reputation categories. The reputation categories are:

```
"Botnet", "Anonymous Proxy", "Phishing", "Spam", "Others", "Tor"
```

If IP string is not a valid IP, return nil.

Return value example:

```
{ "Anonymous Proxy", "Phishing" }
```

ip.geo(“ip-string”) / ip.geo(ip_class)

Return GEO country name in string. If nothing is found or the IP string is not a valid IP, return nil.

ip.geo_code(“ip-string”) / ip.geo_code(ip_class)

Return GEO country code in string. If nothing is found or the IP string is not a valid IP, return nil.

IP address classes

__eq()

Support use “==” to compare two IP address classes.

__tostring()

Support use tostring(IP-class) to convert IP address class to IP string.

str()

Return IP string of this IP address class.

ver()

Return IP address version with integer 4 or 6.

v4()

Return a new IP address class in v4 version. If the IP address class is v4, copy the IP address class and return. If the IP address class is v6, the system will try to convert it to v4. If it succeeds, return the v4 IP address class. If it fails, return nil.

v6()

Return a new IP address class in v6 version. If the IP address class is v6, copy the IP address class and return. If the IP address class is v4, the system will try to convert it to v6. If it succeeds, return the v6 IP address class. If it fails, return nil.

eq(“IP-string”) / eq(IP_class)

Compare this IP address class with another one. It can compare IP address class or IP string.

Predefined commands

All commands are Lua classes but they only can be used inside scripting events. Some commands can only be used in specific events. For example, HTTP commands can only be used inside HTTP events (HTTP_REQUEST and HTTP_RESPONSE).

IP commands

IP commands can be used in HTTP and TCP events.

IP:local_addr()

Return IP address class, which is the local address of the connection.

IP:remote_addr()

Return IP address class, which is the remote address of the connection.

IP:client_addr()

Return IP address class, which is the client IP address of the stream.

IP:server_addr()

Return IP address class, which is the server IP address of the stream. If server is not connected, return nil.

IP:version()

Return the IP version of the connection, either 4 or 6.

TCP commands

TCP commands can be used in HTTP and TCP events.

TCP:local_port()

Return local TCP port of the connection. The value is integer.

TCP:remote_port()

Return remote TCP port of the connection. The value is integer.

TCP:client_port()

Return client TCP port of the connection. The value is integer.

TCP:server_port()

Return server TCP port of the connection. The value is integer. If the server is not connected, return nil.

TCP:close()

Close current TCP connection and disable its TCP events. This function can only be used in event SERVER_CONNECTED.

LB commands

LB commands can be used in HTTP events.

LB:routing(“cr-name”)

Force current HTTP transaction to route to specific content routing.

Return value is Boolean. If the policy doesn’t have content routing or cannot find the specific content routing, return false. If routing successes, return true.

LB:persist(“key”)

LB:persist(“key”, timeout)

Use the key string to do persistence. The type of the server pool’s persistence must be set to scripting, otherwise the function has no effect.

If argument timeout doesn’t exist, use the default timeout in the persistence of the server pool.

If called in HTTP_REQUEST, the system will use the key to search the persistence table. If found, do persistence; If no found, insert key to the persistence table.

If called in HTTP_RESPONSE, the system will insert the key string to the persistence table.

HTTP Commands

HTTP commands can be used in HTTP events.

Header fetch

HTTP:headers()

Fetch all HTTP request or response headers. When it is called in client side, it returns all HTTP request headers; When it is called in server side, it returns all HTTP response headers.

Return: lua table of array.

```
for k, v in pairs(HTTP:headers()) do
   for i = 1, #v do
      debug("HEADER: %s[%d]: %s\n", k, i, v[i])
   end
end
```

HTTP:header(“header-name”)

Fetch specific HTTP request or response header.

Return: lua array

```
for i, v in ipairs(HTTP:header(“set-cookie”)) do
   debug(“set-cookie[%d]: %s\n”, i, v)
end
```

HTTP:cookies()

Fetch all cookies. When it is called in client side, it fetches “Cookies”; When it is called in server side, it fetches “Set-Cookie”.

Return: lua table containing only keys and values.

```
for k, v in pairs(HTTP:cookies()) do
   debug("Cookie: %s = %s\n", k, v)
end
```

HTTP:cookie(“cookie-name”)

Fetch the value of specific cookies.

Return: string.

```
persist = HTTP:cookie(“persist”)
```

HTTP:args()

Fetch all arguments of HTTP query.

Return: lua table containing key and value.

```
for k, v in pairs(HTTP:args()) do
   debug("ARG: %s = %s\n", k, v)
end
```

HTTP:arg(“arg-name”)

Fetch the value of specific arguments.

Return: string.

```
v = HTTP:arg(“ip”)
```

HTTP:host()

Return the string of HTTP request host.

HTTP:url()

Return the string of HTTP request URL. It is full URL including path and query.

HTTP:path()

Return the string of HTTP request path.

HTTP:method()

Return the string of HTTP request method.

HTTP:version()

Return the string of HTTP request or response version.

HTTP:status()

Return two strings including HTTP response status code and reason.

```
code, reason = HTTP:status()
```

Header manipulate

HTTP:set_path(“new-path”)

Change the path in HTTP request header.

Return true for success and false for failure.

```
HTTP:set_path("/new_path")
```

HTTP:set_query(“new-query”)

Change the query in HTTP request header.

Return true for success and false for failure.

```
HTTP:set_query("test=1")
```

HTTP:set_url(“new-url”)

Change the whole URL, including the path and query.

Return true for success and false for failure.

HTTP:set_method(“new-method”)

Change the method in HTTP request header.

Return true for success and false for failure.

```
HTTP:set_url("/new_path?test=1")
```

HTTP:set_status(status-code)

HTTP:set_status(status-code, “reason”)

Change the status code and reason in HTTP response header. If reason does not exist, use default reason.

Return true for success and false for failure.

```
HTTP:set_status(200)
HTTP:set_status(200, "Other Reason")
```

HTTP:add_header(“header-name”, “header-value”)

Add a header line to HTTP request or response header.

Return true for success and false for failure.

Example:

```
function rewrite_request(HTTP, IP, args)
   debug("%s", IP:client_addr())
   client_ip = IP:client_addr()
   -- add/del/set header
   HTTP:add_header("X-COUNTRY-FMF", ip.geo(client_ip) or "unknown") -- add a new
        header line
end
when HTTP_REQUEST {
   local path = HTTP:path()
   if path == "/rewrite_request" then
      rewrite_request(HTTP, IP, HTTP:args())
   end
}
```

HTTP:del_header(“header-name”)

Remove the header with name “header-name” from HTTP request or response.

Return true for success and false for failure.

HTTP:set_header(“header-name”, header-value-array)

Remove the header with name “header-name” from HTTP request or response, and add this header with new value header-value-array. The argument header-value-array is a Lua array which is the value got from HTTP:header().

Return true for success and false for failure.

```
HTTP:set_header("test", { "line1", "line2", "line3" })
```

HTTP:replace_header(“header-name”, “regex”, “replace”)

Match the regular expression in all occurrences of header field “header-name” according to “regex”, and replaces them with the “replace” argument. The replacement value can contain back references like 1,2, …

Return true for success and false for failure.

```
-- add api to set-cookie path
HTTP:replace_header("set-cookie", [[(.*)(Path=\/)(.*)]], [[\1\2api\3]])
```

Custom reply

These functions only can be used in HTTP client side event (only HTTP_REQUEST now).

HTTP:redirect (“fmt”, …)

Reply to client with redirect response.

```
HTTP:redirect(“https://%s”, HTTP:host())
```

HTTP:reply (response)

Reply to client with custom response.

Argument response is a lua array. It includes:

- status: Integer. Default is 200.
- reason: String. If not set, the system will use the default value of status code. For example, if the status code is 200, the default value of reason is “OK”.
- headers: Lua table. Each value of the table is a lua array. It contains all headers except “content-length”. “content-length” will be automatically set with the body size.
- Body: String.

```
HTTP:reply{
   status = 400,
   reason = “test reason”,
   headers = {
      ["content-type"] = { "text/html" },
      ["cache-control"] = { "no-cache", "no-store" },
```

```
    },
    body = "<html><body><h1>invalid request<h1></body></html>",
}
```

Control

HTTP:close()

Close the current HTTP transaction and disable its HTTP events. This function can only be used in event HTTP_REQUEST.

Protocol

HTTP:is_https()

Return true if the current transaction is in HTTPS connection.

Transaction private data

In Lua, the local value can only be used in function and the global value is shared in whole Lua stack.

In FortiWeb, sometimes a private data is needed for HTTP transaction, and the value is shared in the same HTTP transaction.

HTTP:setpriv(object)

Store a lua object as the HTTP transaction private data. You can store a lua object in event HTTP_REQUEST and fetch it by calling HTTP:priv() in event HTTP_RESPONSE.

HTTP:priv()

Fetch the transaction private data that stored by HTTP:setpriv(). If no result is found, it will return an empty lua table.

Data Collect

HTTP:collect()

This function only exist in script events HTTP_REQUEST and HTTP_RESPONSE.

Example

```
when HTTP_REQUEST {
--HTTP:collect command can be used in both HTTP_REQUEST and HTTP_RESPONSE events
--size, if size is -1 it will collect up to the full length or when FortiWeb's max-
     cahched length is reached
if HTTP:header("content-type") == text/css
HTTP::collect()
}
```

Body Rewrite

HTTP:body (offset, size)

Offset and size are optional.

If offset is missing, it will be set as zero.

If size is missing, it will be set as -1 which means the whole HTTP body.

Return string. The HTTP body will be returned.

HTTP:set_body("body_str", offset, size)

Offset and size are optional.

If offset is missing, it will be set as zero.

If size is missing, it will be set as -1 which means the whole `http` body.

The body_str is the HTTP body. Now only the string type body is supported.

Return boolean: `true/false`.

Example of HTTP replace body

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

--This function will change "username:test" to "username:Test"
function username_first_char_uppercase(str)
local str1 = str:sub(1, 9)
local str2 = str:sub(10, 10)
str2 = str2:upper()
local str3 = str:sub(11, -1)
return str1..str2..str3
end

when HTTP_DATA_REQUEST {
local body_str = HTTP:body(0, 16)
local body_new = body_str:gsub("username:[A-Za-z][A-Za-z0-9_]+", username_first_char_
     uppercase)
debug("body old = %s, body new =  %s\n", body_str, body_new)
HTTP:set_body(body_new, 0, 16)
}
```

www.fortinet.com