TRISUL NETWORK ANALYTICS

http://trisul.org

LUA API Reference Documentation

version 4.0

Table of Contents

1 Introduction....3
1.1 LUA version....3
1.2 Installing Lua scripts and error handling....3
1.2.1 Loading and error handling....3
1.3 Basic structure of a LUA script....4
1.4 On GUIDs....5
1.5 Top level functions onload() and onunload()....5
1.6 The ID Block....5
2 Globals T....6
2.1.1 T.host methods....6
2.1.2 T.K methods....7
T.K.vartype....7
2.1.3 T.util methods....8
2.1.4 T.debugger....9
2.1.5 T.re2....10
2.1.6 T.ac....11
3 Objects....12
3.1.1 Engine....12
3.1.2 Layer....14
3.1.3 Packet....15
3.1.4 Buffer....15
3.1.5 FlowID....16
3.1.6 RE2....17
3.1.7 AC....18
4 Script types....19
4.1 Counter Group....20
4.1.1 Table countergroup....20
The countergroup > control table....21
The countergroup > meters table....21
The countergroup > keyinfo table....22
Format....22
4.2 Simple Counter....23
4.2.1 Table simplecounter....23
4.2.2 The simplecounter > onpacket function....24
The layer parameter....24
The engine parameter....25
4.3 Flow Monitor....26
4.3.1 Table flowmonitor....26
List of supported flow attributes....27
4.3.2 Function flowmonitor > onflowattribute....28

1 Introduction

1.1 LUA version

Trisul embeds LuaJIT which is compatible with Lua version 5.1.

1.2 Installing Lua scripts and error handling

LUA scripts are contained a single file, say `myplugin.lua`.

- To install the script just place it in the `/usr/local/lib/trisul/plugins/lua` directory. They are automatically picked up when you restart Trisul.
- To uninstall, remove them from the above directory.
- You need to restart Trisul to make it load new scripts.

1.2.1 Loading and error handling

All `.lua` files in the `plugins/lua` directory will be inspected for capabilities by the Trisul runtime. If there are any syntax errors, the scripts will simply fail to load with no output on the command line. The errors can be found in the main Trisul log file instead. The log files are located in `/usr/local/var/log/trisul/ns*.log`

Some techniques

1. All messages will contain the filename of your Lua script, so you can grep `myfile.lua`
2. You can also try loading it outside of Trisul via a Lua command line to rule out syntax errors

A typical error message looks like this

```
# grep re2http.lua /usr/local/var/log/trisul/ns*.logSat Apr  5 16:27:49
2014.048575 ERROR [re2http.lua]Unable to load lua file, see next message

Sat Apr  5 16:27:49 2014.048588 ERROR [re2http.lua]LUA file error :
plugins/lua/re2http.lua:51: ')' expected (to close '(' at line 50) near 'for'
```

1.3 Basic structure of a LUA script

A LUA script follows the following structure

```
TrisulPlugin = {
  id = {
    ...
  }
  onload = function()
    ...
  end
  onunload = function()
    ...
  end
  -- plugin type
  -- here we select simplecounter in this example
  simplecounter = {
    ...
  }
}
```

You may also use the other LUA notations such as the following.

```
TrisulPlugin.onload = function()
  ..
end..
```

1.4 On GUIDs

GUIDs are used throughout the scripting interface. A GUID is just a globally unique ID. To avoid namespace issues Trisul uses a GUID to identify each counter group, protocol, and many other things. For more on GUIDs and for a list of common GUIDs refer to Common GUIDs

1.5 Top level functions onload() and onunload()

These methods, if present, are called for every plugin.

onload	optional	Called when the script is loaded into Trisul. You may do any initialization such as reading datafiles here
onunload	optional	Called when the script is unloaded. Free up resources

One important point is your script may be loaded and unloaded several times by Trisul. Also more than one instance of your script may be loaded and active. Do not make any assumptions about the singleton nature of your scripts.

1.6 The ID Block

Every plugin must have an ID block with the following fields

name	string	Mandatory: A short name for the script
description	string	Optional (default = none) : More info about the plugin
author	string	Optional (default = Unleash) : Who wrote the script
version_major	number	Optional (default = 1) : A major version number
version_minor	number	Optional (default = 0): A minor version number

2 Globals T

The global table ***T*** can be accessed from anywhere. The following methods are available on T.

T.host	Host methods that can be called from LUA
T.K	Constants
T.utils	Utility methods
T.debugger	Drops you into an interactive debugger when called
T.re2	A fast and powerful regex engine (Google RE2)
T.ac	A minimal but fast Aho-Corasick multi pattern matcher

2.1.1 T.host methods

Interact with the Trisul environment.

Name	In	Out	Description
log	level, msg		Log a message to the Trisul log file, usually located in /usr/local/var/log/trisul Usage:`T.host:log(T.K.loglevel.INFO, "HELLO ")`
get_homenets		Table, Array of [IP, Netmask]	Get home networks defined by Trisul.
is_homenet	32-bit	bool	Is the 32-bit IPv4 address within the home network?
get_configpath		string	Configuration directory
get_datapath		string	Data directory
createkey	cgguid, key, label		Create a userlabel for a given key. Use this to pre-load human labels for keys
prepare_config			Prepare a configuration file for your plugin if needed
broadcast			Broadcast a state update to other plugins.

2.1.2 T.K methods

Pre-defined constants to use with other Lua functions.

loglevel	Log levels	EMERG,FATAL,ALERT,CRIT,ERROR,WARN,NOTICE,INFO,DEBUG Usage :print(T.K.loglevel.ERROR) => 4
vartypes	Meter types	COUNTER, RATE_COUNTER,GAUGE, RUNNING_COUNTER

T.K.vartype

Constants : Types of counters.

COUNTER	Increment a counter that resets to zero at start of every time bucket
RATE_COUNTER	Equal to COUNTER/Bucket Size in seconds
GAUGE	Instantaneous values
RUNNING_COUNTER	Increment or decrement a counter, does not reset every time bucket

2.1.3 T.util methods

Some utility functions.

ntop	32-bit number	string	Convert a 32 bit number to IPv4 address string
pton	string	number	Convert an IPv4 address string to a number
bor	number,number	number	bitwise OR of two numbers
band	number,number	number	Bitwise AND of two numbers
testbit32	number,number(bit position)	bool	Test bit position of a 32 bit number.LSB=0,MSB=31 T.util.testbit32(num,8)@ returns true if bit 8 = 1
bitval32	number, number (start bit), number (width)	number	Get value of continous bits. `T.util.bitval32(num,20,4)` returns the numeric value of bits 20,19,18,17

bit32 in Lua 5.2 note The bit utilities have been provided because we are using LuaJIT which does not support Lua5.2's bit32 library.

2.1.4 T.debugger

T.debugger is an interactive debugger that you can call. When called, this method drops you into an interactive LUA shell where you may run one line commands. This lets you play with the objects, strings, protocol bits to help you build your script.

A sample session

Running this code

```
onpacket = function(engine,layer)
local buff = layer:rawbytes()
 ...


T.debugger({ engine = engine, layer = layer })
```

Blocks the packet processing pipeline and drops you into a shell as shown below.

```
Trisul LUA [0x8fc750]> print(layer:layer_bytes())
20
Trisul LUA [0x8fc750]> print(layer:rawbytes():hexdump())
00000000   00 16 0b 94 c1 e9 e6 b3 06 7c e0 ce 50 10 7f ff    .........|..P...
00000010   07 85 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
Trisul LUA [0x8fc750]>
```

Parameters

T.debugger	table	A table containing symbols you want to export to the debugger sandbox. `T.debugger( {e = engine, l = layer})` will allow your to use the symbols `e` and `l` in your debugger CLI

2.1.5 T.re2

Google RE2 is a fast, threadsafe, and powerful Regex engine. Trisul exposes useful methods that are most frequently used in matching and extracting info from headers in network protocols.

1. T.re2 also allows you to employ very common string matching idioms like `(octet-stream |application-x | application-pdf)`which arent available in Lua's find method.
2. T.re2 allows you to precompile the regexes once and run them later

T.re2	string	A re2 object	Precompile the regex string and return an re2 object

A sample illustrating a typical use.

```
-- precompile in onload

 onload = function()

   my_regex = T.re2("User-Agent\\s*:\\s*(.*)\r\n")

  end

  onflowattribute = function(...)

    --

    -- my_regex is a precompiled re2 object, just compare

    -- or extract from a target string

    ---

    if my_regex:partial_match( a_string ) then

      ..

    end
```

2.1.6 T.ac

Multi-pattern matching is such a common use case for our purposes that we have provided a minimal but fast Aho-Corasick multi pattern matcher.

T.ac	table (An array of patterns)	An AC object	Load all the patterns into an AC matcher and return an AC matcher object

A sample illustrating a typical use.

```
onload = function()

    -- add patterns in array and create a new AC matcher

    ac_headers   = T.ac({ "Host:",

                          "User-Agent:",

                          "Referer",

                          "Server:",

                          "Content-Type:",

                          "Content-Length:"} )

   -- later on you can use the match methods

   ac_headers:match_all(...)
```

3 Objects

The following objects are available. The objects are implemented as metatables in C, so in order to call a function on an object use the following syntax.

```
object:method(..)
```

3.1.1 Engine

The Trisul stats engine.

Name	In	Out	Description
update_counter	guid, key, meter, val		Updates a meter for a key with a particular value
update_counter_bytes	guid, key, meter		Updates a meter for a key but with a value of wire_length in the packet
add_alert	guid, flowkey, alertkey, details		Add an alert GUID = an alert group flowkey = a flow id say from flowID:id alertkey = a SIGID details = a message
timestamp			tv_sec and tv_usec/nsec. The latest timestamp seen by the engine
add_flow_counter	flowkey, guid, key, meter	number, number	Add a flow counter. Automatically counts the guid,meter,key tuple for each subsequent packet in the flow
reset_flow_counter	flowkey, guid, key, meter		Removes all flow counters, then does an add_flow_counter
add_alert_full	guid, flowkey, alertkey, class, priority, details		Same as add alert with with a priority and classification to make it integrate better with other IDS alerts. Use the format `sn-1` for priority 1
tag_flow	flowkey, tag		Tag a flow with a label. You can then

Name	In	Out	Description
			search for flows with that label The labels should be SHORT < 10 characters to work will in the web UI

3.1.2 Layer

Packet contents at a given protocol layer.

Name	In	Out	Description
layer_bytes		number	Number of bytes at this layer. Example: UDP layer has 8 bytes
protocol_id		GUID	The GUID of the protocol
rawbytes		A Buffer object	Raw bytes
total_bytes		number	Total number bytes in the packet, i.e. same as Packet:wire_length()
packet		A Packet object	The full packet
testbit	bitno	bool	Test if a bit is set in this layer. Bits are numbered serially from *0 to 8*layer_bytes-1* This is a shortcut for `layer:rawbytes():hval_8(bitno/8)` You are encouraged to use `testbit` because it increases performance by reducing Lua-C round trips.
getbyte	offset	number	Get a byte at this offset. Shortcut for `layer:rawbytes():hval_8(offset)` You are encouraged to use `getbyte` because it increases performance by reducing Lua-C round trips.

3.1.3 Packet

Represents a packet.

This contains layers such as Ethernet/IP/UDP etc.

Name	In	Out	Description
timestamp		number, number	two numbers representing tv_secs and tv_usecs (or nsecs)
rawbytes		A Buffer object	the full packet
wire_length		number	packet length
capture_length		number	number of bytes captured. Could be less than wire_length due to a snaplen setting
num_layers		number	number of layers
get_layer	number	A Layer object	get a layer by index Note: Index starts from 0..num_layers-1 ; unlike LUA
find_layer	guid	A Layer object	get layer identified by the GUID

3.1.4 Buffer

Raw bytes.

Name	In	Out	Description
size		number	size of the buffer containing the raw bytes
hexdump	offset, size (both optional)	string	string with hexdump of the buffer in canonical format. Specify offset, size to dump a subset
hval_8	offset	number	returns the byte at buffer+offset
hval_16	offset	number	same as *ntohs( )* on the 2 bytes at buffer+offset
hval_24	offset	number	The 3 byte (24 bit) number at buffer_offset
hval_32	offset	number	same as *ntohl( )* on the 4 bytes at buffer+offset
tostring	offset, size (both optional)	string	`tostring()` returns the full string to LUA `tostring(offset,size)` returns the substring of

Name	In	Out	Description
			size bytes starting and including *offset*

3.1.5 FlowID

Represents a flow.

Name	In	Out	Description
id		string	A unique string identifying the flow
protocol		string	IP protocol, TCP/GRE/UDP/etc
ipa		string	*Trisul Key Format* : IP Address of A-End can be IPv4 or IPv6
ipa_readable		string	*Human readable format* : IP Address of A-End can be IPv4 or IPv6
porta		string	*Trisul Key Format* : port
porta_readable		string	*Human readable format* : port number
ipz		string	*Trisul Key Format* : IP Address of A-End can be IPv4 or IPv6
ipz_readable		string	*Human readable format* : IP Address of A-End can be IPv4 or IPv6
portz		string	*Trisul Key Format* : port
portz_readable		string	*Human readable format* : port number
netflow_router		string	Netflow router ID
netflow_ifindex_in		string	Netflow input ifIndex
netflow_ifindex_out		string	Netflow output ifIndex

3.1.6 RE2

A precompiled regex created with `T.re2(_expression_)`

Name	In	Out	Description
partial_match	string	bool	Does the regex match anywhere in the input string
full_match	string	bool	Full input string must match the regex
partial_match_c1	string	bool, string	Match input string with 1 capture. If matched, return true + captured string Example `(Host.*)\\s*:` will pull the match in paranthesis into a string. If not matched, return false, nil.
partial_match_c2	string	bool, string, string	Same as partial_match_c1 but extract TWO captures rather than one.

More will be added as required. For RE2 syntax which is a bit different from PCRE visit the Google Re2 page

3.1.7 AC

An Aho-Corasick multi pattern matcher created with `T.ac( _pattern_array_ )`

Name	In	Out	Description
match_all	string	table	Matches all patterns. The matches are returned in a table { pattern_matched = position } The position indicates the last matching character, not the first.
match_one	string	table	Same as match_all, but stops after finding a single match. Use this method for alerting on pattern matches.

4 Script types

The following types of LUA scripts are supported.

A single LUA file may contain multiple script types, or each script type could be in its own LUA file.

Name	Called When	Notes
countergroup	During initialization	Create a new counter group
simplecounter	Called for each packet	Inspect packet contents and update meter/alerts
flowmonitor	Called when interesting data is seen per flow	Inspect HTTP headers, TLS certs, then attach counters to flows

We describe each type of script in detail in the sections below.

4.1 Counter Group

Creates a new counter group and associated meters.

> Requires a LUA table named `countergroup`

4.1.1 Table countergroup

The table countergroup has two sections called `control` and `meters`. The control section describes the counter group and the meters section define the individual meters within the group. The countergroup table typically looks like this.

```
countergroup = {
  control = {
    ...
    -- id and parameters of new group
  },
  meters = {
    ...
    -- info about the various meters
  },
  keyinfo = {
    ...
    -- keys to label mappings
  },
}
```

The countergroup > control table

The control table assigns a unique GUID to the counter group and attaches it to a particular layer in the network protocol stack.

guid	string	A unique guid that identifies the group. See section on GUIDs
name	string	Name of the counter group. Keep it short < 15 chars
description	string	
bucketsize	number	Resolution of the counter group for all meters in seconds.

The countergroup > meters table

Every counter group can house upto 16 different meters. This section defines each of them.

The meters section is an “array of arrays”, or in the LUA world, a “table of tables”. The typical format is the following

```
meters = {
    {  0, T.K.vartype.RATE_COUNTER, 10, "Bytes",   "bytes" ,   "B" },
    {  1, T.K.vartype.COUNTER,      10, "Packets", "packets",  "Pkts" },
    ..
},
```

Each meter line defines the following

0 Meter ID must start from 0

1 Vartype type type of meter, see the global named T.k.vartype

2 Top counter how many toppers do you want to track for the this meter

3 Name Meter name (keep it short < 10 charts)

4 Description what does it track

5 Units Suffix for units, must be compatible with K, M, G for Kilo, Mega, Giga etc

The countergroup > keyinfo table

Each entity being monitored in a counter group is identified by a *key* string. You are responsible for creating these key strings. The *keyinfo*table maps these keystrings into user friendly display labels. The web UI shows these labels instead of the raw keys.

Format

The keyinfo “table” is an array of `{ key, label }`

key string

label string

```
keyinfo={
  {"14/00","change_cipher_spec"},
  {"15/00","alert"},
  {"16/00","hello_request"},
 ..
}
```

4.2 Simple Counter

Inspect each packet to create your own metering and alerting logic.

4.2.1 Table simplecounter

The simplecounter table attaches the onpacket(..) function to a particular protocol layer. The two attributes of the table are:

protocol_guid	string	Which protocol do you want to attach this group to? Trisul will invoke this counter group only for packets where that protocol is present and with a pointer to the payload at that protocol A list of common protocol GUIDs are here
onpacket	function	`onpacket(..)` is the main function where your LUA code goes. It is called for each packet with a layer object corresponding to the protocol_guid above

A typical simplecounter table looks like the following

```
simplecounter={

  protocol_guid = "{0A2C724B-5B9F-4ba6-9C97-B05080558574}",

  onpacket = function(engine, layer) {

    -- your lua code goes here

    ...

 },

}
```

4.2.2 The simplecounter > onpacket function

The onpacket function is where your LUA code goes. You can inspect the packet bytes, then apply your own logic and interact with the Trisul engine.

engine	An Engine object that allows you to interact with Trisul
layer	A Layer object pointing to the `protocol_guid` you have specified

The `layer` parameter

This code dumps the IP header and the size of the IP layer. Once again the GUID "{0A2C724B-5B9F-4ba6-9C97-B05080558574}" represents the IPv4 protocol

```
simplecounter = {

  protocol_guid = "{0A2C724B-5B9F-4ba6-9C97-B05080558574}",

  onpacket = function(engine,layer)

      print("onpacket now.. layer length = "..layer:layer_bytes())

      print("Hexdump\n")

      local bytes = layer:rawbytes()

      print(bytes:hexdump())

  end,

},
```

The above snippet works as follows

- print the IP layer length via `layer:layer_bytes()` See Layer
- dump the 20 byte IP header using `layer:rawbytes():hexdump()`

```
onpacket now.. layer length = 20

Hexdump

00000000   45 00 00 28 a3 c8 40 00 35 06 15 4a d1 d8 f9 3a    E..(..@.5..J...:

00000010   c0 a8 01 02 00 00 00 00 00 00 00 00 00 00 00 00    ................

..
```

The `engine` parameter

Typically the end result of your LUA processing will result in a call to one of the Engine methods. Check out the samples for how they are used.

4.3 Flow Monitor

Attach counters to flows based on content.

4.3.1 Table `flowmonitor`

The table contains a single function called `onflowattribute` The function is passed the following parameters

engine	An Engine object
flow	A Flow ID object
timestamp	a timestamp in epoch_secs
attribute_name	See below
attribute_value	Value of the said attribute. It is a Buffer object

Essentially, the way flow monitors work are as follows.

1. Trisul extracts bits of information from flows called “flow attributes”
2. Examples of flow attributes are HTTP headers, URLs, Content Types TLS Certificates, TLS organizations, handshake records, etc
3. You can observe these flow attributes and attach a counter to the flow
4. When you attach a counter to a flow, all packets in that flow are automatically counted without calling your LUA plugin for every packet

List of supported flow attributes

The flow attributes currently supported are

String	Description
“”	A blank (empty string “”) attribute type and value is sent whenever a new flow is started, you can use the Flow IDmethods to get the flow details
^D	A “^D” (caret followed by capital D) A flow has terminated or timeout.
HTTP-Header	Complete HTTP header both requests and responses
URI	Complete URL including query params
Content-Type	Just the Content-Type found in HTTP responses
User-Agent	From the HTTP Header
Host	From the HTTP Header
TLS:RECORD	The full TLSrecord. All TLS Content_types except application_data (23) are sent down to LUA
TLS:O	TLS organization , ie subject found in a certificate
TLS:CA:ROOT	TLS Root CA
TLS:CA:INTER	TLS Intermediate CA names
TLS:CIPHER	TLS Ciphersuites
more will be added	coming up..

4.3.2 Function flowmonitor > onflowattribute

The function onflowattribute is designed like a **filter**. It is called for ALL flow attributes, even those you may not be interested in. You have to use a `if attribute_name == ".."` construct to filter out noise you are not interested in.

Lets say you only want to deal with `HTTP-Header`- you would do something like below

```
flowmonitor = {


  onflowattribute =

       function(engine,flow,timestamp,attribute_name,attribute_value)

          if attribute_name == "HTTP-Header" then


             -- do your thing here

             local hdr = attribute_value:tostring()

             ...

          end

       end

}
```

If you're keeping track of some state information per flow, you can use the `"^D"` message to cleanup.