

Lua Scripting in Wireshark

June 17, 2009

Stig Bjørlykke

Software Troubleshooter | Thales Norway AS

SHARKFEST '09

Stanford University

June 15-18, 2009

THALES

Introduction

- About me
 - I'm working as a senior system developer for Thales Norway, a company focusing on defence, aerospace and security markets worldwide
 - Wireshark user since 2003
 - Wireshark core member since 2007
 - I enjoy parachuting and scuba diving

Agenda

- Introduction to Lua
 - Getting started using Lua in Wireshark
- Functions to write a dissector
 - Obtaining dissection data
 - Presenting information
 - Preferences
 - Post-dissectors
- Functions to create a Listener

Introduction to Lua



- Lua is a powerful, fast, lightweight, embeddable scripting language designed for extending applications.

Introduction to Lua

- Script language
 - Good support for object-oriented programming
- Can be precompiled for
 - Faster loading (not faster *execution*)
 - Off-line syntax error detection
 - Protecting source code from user changes
- Lua's official web site
<http://www.lua.org/>

Lua variables

- Dynamically typed language
- All values are *first-class values*
- Eight basic types
 - nil, boolean, number, string, function, userdata, thread and table
- All variables are global unless using the *local* keyword

Lua in Wireshark

- Usage in Wireshark
 - Dissectors
 - Used to decode packet data
 - Post-dissectors
 - Called after every other dissector has run
 - Listeners
 - Used to collect information after the packet has been dissected

Lua in Wireshark

- Advantages
 - Easy prototyping, implementing and testing
 - Small amount of code needed
 - No memory management
 - Easy to share with others
 - Perfect for reverse engineering

Lua in Wireshark

- Disadvantages
 - Several times slower than writing in C
 - Only a subset of dissector functions
 - Code is not distributed with Wireshark
 - Not widely used yet

Lua in Wireshark

- How Lua fits into Wireshark
 - A file called `init.lua` will be called first
 - First from the global configuration directory
 - Second from the personal configuration directory
 - Scripts passed with the `-X lua_script:file.lua` will be called after `init.lua`
 - All scripts will be run **before** packets are read, at the end of the dissector registration process.

Lua in Wireshark

- Not fully implemented yet
 - Not built by default on all platforms
 - Disabled in the init scripts
 - Still missing some functionality
 - Documentation is incomplete
 - Few working examples available
 - Probably still some bugs

Getting started

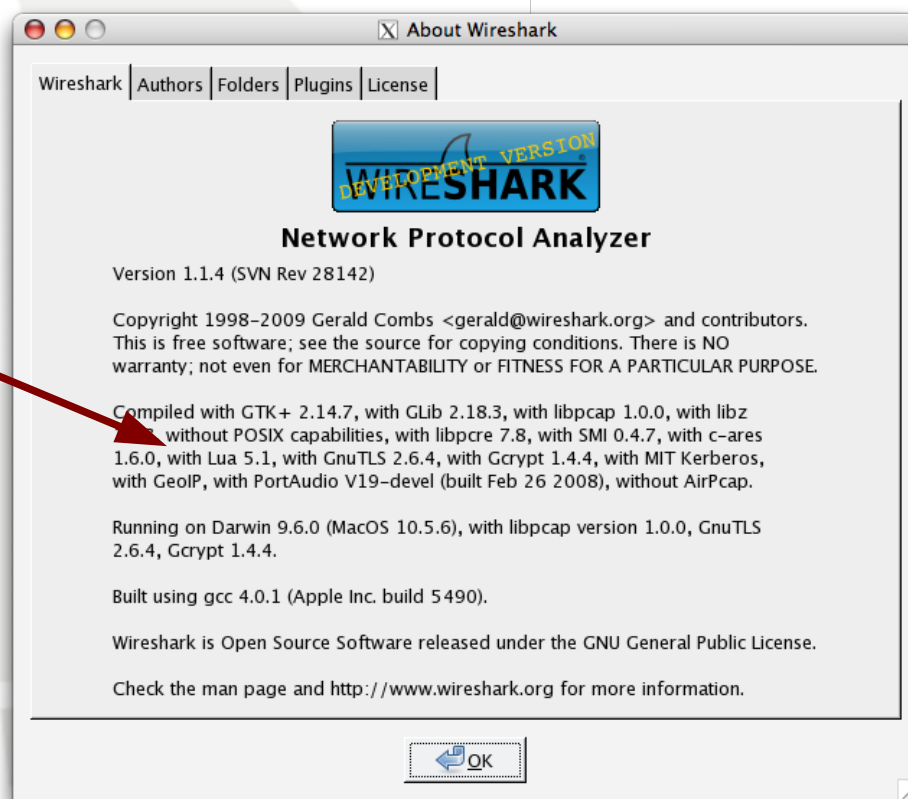
1. Check your version of Wireshark

Help -> About

Compiled with GTK+ 2.14.1.2.3, without POSIX capabilities, with Lua 5.1, with Glib 2.16.0, with GeoIP, with PortAudio

versus

Compiled with GTK+ 2.12.1.2.3, without POSIX capabilities, with ADNS, without Lua, with Glib 2.16.0, with PortAudio v19-devel (built



Getting started

2. Enable LUA in the global configuration file

Remove the `disable_lua` line from `init.lua`

File can be found from:

Help -> About -> Files -> Global configuration

```
-- Lua is disabled by default, comment out the following line
-- to enable Lua support.
disable_lua = true; do return end;

-- If set and we are running with special privileges this setting
-- tells whether scripts other than this one are to be run.
run_user_scripts_when_superuser = false
```

Getting started

3. Create a test script to check if it works

```
-- hello.lua  
-- Lua's implementation of D. Ritchie's hello world program.  
print ("Hello world!")
```

Getting started

4. Test the hello.lua script

This can be done with `tshark`

```
$ tshark -X lua_script:hello.lua
```

```
Hello world!
```

```
Capturing on AirPort
```

```
1 0.000000 192.168.1.55 -> 192.156.1.255 NBNS Name query NB XXX.COM<00>
```

Create a simple dissector

- Example: My Simple Protocol
 - Protocol specifications
 - Message Id (4 bytes)
 - Magic Value (4 bits)
 - Message Format (4 bits: 1=Text 2=Binary)
 - Data (variable length)
 - Runs on UDP port 1000

Create a new protocol

- Proto
 - Creates a new protocol in Wireshark
 - proto.dissector: a function you define
 - proto.fields: a list of fields
 - proto.init: the initialization routine
 - proto.prefs: the preferences
 - proto.name: the name given

Create a new protocol

```
-- Create a new dissector
```

```
MYPROTO = Proto ("myproto", "My Simple Protocol")
```

Add a protocol dissector

- Proto.dissector
 - This is the function doing the dissecting
 - Takes three arguments: buffer, pinfo and tree

```
-- The dissector function  
function MYPROTO.dissector (buffer, pinfo, tree)  
    <do something>  
end
```

Create protocol fields

- ProtoField
 - To be used when adding items to the tree
 - Integer types:
 - ProtoField.{type} (abbr, [name], [desc],
[base], [valuestring], [mask])
uint8, uint16, uint24, uint32, uint64, framenum
 - Other types
 - ProtoField.{type} (abbr, [name], [desc])
float, double, string, stringz, bytes, bool, ipv4, ipv6, ether,
oid, guid

Create protocol fields

- Proto.fields
 - Contains a list of all ProtoFields defined

```
-- Create the protocol fields
```

```
local f = MYPROTO.fields
```

```
local formats = { "Text", "Binary", [10] = "Special"}
```

```
f.msgid   = ProtoField.uint32 ("myproto.msgid", "Message Id")
f.magic    = ProtoField.uint8  ("myproto.magic", "Magic", base.HEX, nil, 0xF0)
f.format   = ProtoField.uint8  ("myproto.format", "Format", nil, formats, 0x0F)
f.mydata   = ProtoField.bytes  ("myproto.mydata", "Data")
```

The protocol initialization

- Proto.init
 - Called before we make a pass through a capture file and dissect all its packets
 - E.g. when we read in a new capture file, or run a «filter packets» or «colorize packets»

```
-- A initialization routine  
local packet_counter  
function MYPROTO.init ()  
    packet_counter = 0  
end
```

Fetch data from the packet

- Tvb / TvbRange
 - The buffer passed to the dissector is represented by a tvb (Testy Virtual Buffer)
 - Data is fetched by creating a TvbRange
 - Tvb ([offset], [length])
 - The tvbrange can be converted to correct datatypes with this functions
 - uint, le_uint, float, le_float, ipv4, le_ipv4, ether, string, bytes

Fetch data from the packet

```
-- The dissector function

function MYPROTO.dissector (buffer, pinfo, tree)

  -- Fetch data from the packet
  local msgid_range = buffer(0,4)
  local msgid = msgid_range:uint()

  -- This is not supported in Wireshark, yet
  local format = buffer(4,1):bitfield(4,4)

  local mydata = buffer(5):bytes()

end
```

Adding fields to the tree

- TreeItem
 - Used to add a new entry to the packet details, both protocol and field entry
 - Adding a new element returning a child
 - treeitem:add ([field | proto], [tvbrange], [label])
 - Modifying an element
 - treeitem:set_text (text)
 - treeitem:append_text (text)
 - treeitem:add_expert_info ([group], [severity], [text])
 - treeitem:set_generated ()

Adding fields to the tree

```
-- The dissector function

function MYPROTO.dissector (buffer, pinfo, tree)

  -- Adding fields to the tree

  local subtree = tree:add (MYPROTO, buffer())
  local offset = 0

  local msgid = buffer (offset, 4)
  subtree:add (f.msgid, msgid)
  subtree:append_text ("Message Id: " .. msgid:uint())
  offset = offset + 4

  subtree:add (f.magic, buffer(offset, 1))
  subtree:add (f.format, buffer(offset, 1))
  offset = offset + 1

  subtree:add (f.mydata, buffer(offset))

end
```

▼ My Simple Protocol, Message Id: 70213

Message Id: 70213

0001 = Magic: 0x01

.... 0010 = Format: Binary (2)

Data: 0100000010000000000000000377777710676F6F676C652D616E...

Register the protocol

- DissectorTable
 - This is a table of subdissectors of a particular protocol, used to handle the payload
 - DissectorTable.get (tablename)
 - The most common tablenamees
 - TCP and UDP uses port numbers
 - «tcp.port» and «udp.port»
 - Ethernet uses an ether type
 - «ethertype»

Register the protocol

```
-- Register the dissector  
  
udp_table = DissectorTable.get ("udp.port")  
udp_table:add (1000, MYPROTO)
```

Packet information

- Read only
 - pinfo.number: packet number
 - pinfo.len: packet length
 - pinfo.rel_ts: time since capture start
 - pinfo.visited: true if package has been visited
- Generated during capture

Packet information

- Read write
 - pinfo.cols: packet list columns
 - pinfo.src
 - pinfo.src_port
 - pinfo.dst
 - pinfo.dst_port
- Generated while dissecting

Modifying columns

- All columns can be modified
 - Most common is protocol and info
 - pinfo.cols.protocol
 - pinfo.cols.info
 - Others can be the addresses
 - pinfo.cols.src
 - pinfo.cols.dst
 - pinfo.cols.src_port
 - pinfo.cols.dst_port

Modifying columns

```
-- The dissector function

function MYPROTO.dissector (buffer, pinfo, tree)

  local offset = 0
  local msgid = buffer(offset, 4)

  -- Modify columns

  pinfo.cols.protocol = MYPROTO.name
  pinfo.cols.info = "Message Id: "
  pinfo.cols.info:append (msgid:uint())

  <continue dissecting>

end
```

No. .	Time	Source	Destination	Protocol	Info
1	0.000000	192.168.39.109	192.168.39.245	MYPROTO	Message Id: 162
2	0.030561	192.168.39.245	192.168.39.109	MYPROTO	Message Id: 162
3	12.100564	192.168.39.64	192.168.39.245	MYPROTO	Message Id: 69
4	12.131395	192.168.39.245	192.168.39.64	MYPROTO	Message Id: 69

Adding preferences

- Pref
 - Creates a preference to be put in Proto.prefs
 - Several types available
 - Pref.{bool,uint,string} (label, default, desc)
 - Pref.enum (label, default, desc, enum, radio)
 - Pref.range (label, default, desc, range, max)
 - Pref.statictext (label, desc)
 - Can be used as a regular variable

Adding preferences

```
-- Add a integer preference

local p = MYPROTO.prefs
p.value = Pref.uint ("Value", 0, "Start value for counting")


-- Use the preference

if not pinfo.visited and msgid:uint() >= p.value then
    packet_counter = packet_counter + 1
end
```

Adding preferences

```
-- Add a enum preference

local p = MYPROTO.prefs

local eval_enum = { { "First", "First value", 0 },
                    { "Second", "Second value", 1 },
                    { "Third", "Third value", 2 } }

p.value = Pref.uint ("Value", 0, "Start value for counting")
p.eval = Pref.enum ("Enum Value", 1, "Another value", eval_enum, true)
p.text = Pref.statictext ("The enum value is not yet implemented")
```

My Simple Protocol

Value:

Enum Value: ☐ First value ☒ Second value ☐ Third value

The enum value is not yet implemented

Create a post-dissector

- A post-dissector is just like a dissector
 - Register a protocol (with a dissector)
 - register_postdissector (Proto)
 - It will be called for every frame after dissection

```
-- Create a new postdissector  
MYPOST = Proto ("mypost", "My Post Dissector")  
function MYPOST.dissector (buffer, pinfo, tree)  
    <do something>  
end  
register_postdissector (MYPOST)
```

Create a Listener

- A Tap is a listener which is called once for every packet that matches a certain filter or has a certain tap.
 - Register a new listener
 - `Listener.new ([tap], [filter])`
 - Must have this functions
 - `listener.packet`
 - `listener.draw`
 - `listener.reset`

Create a Listener

```
-- My Simple Listener

local function my_simple_listener ()
    local tw = TextWindow.new ("My Simple Listener")
    local tap = Listener.new (nil, "myproto")

    tw:set_atclose (function () tap:remove() end)

    function tap.packet (pinfo, buffer, userdata)
        -- called once for every matching packet
    end

    function tap.draw (userdata)
        -- called once every few seconds to redraw the gui
    end

    function tap.reset (userdata)
        -- called at the end of the capture run
    end

    retap_packets ()
end

register_menu ("My Simple Listener", my_simple_listener, MENU_TOOLS)
```

Obtain field values

- Field
 - Fields can be extracted from other dissectors
 - Field.new (filter)
- FieldInfo
 - An extracted Field used to retrieve values
 - fieldinfo.value
 - fieldinfo.len
 - fieldinfo.offset

Obtain field values

```
-- Register a field value

udp_len_f = Field.new ("udp.length")

local function menuable_tap ()
    function tap.packet (pinfo, buffer, userdata)
        -- Fetch the UDP length
        local udp_len = udp_len_f()

        if udp_len and udp_len.value > 400 then
            -- Do something with big UDP packages
        end
    end
end

end
```

Calling other dissectors

- Dissector
 - A reference to a dissector, used to call a dissector against a packet or a part of it.

```
-- Send data to the UDP dissector
```

```
udp_dissector = Dissector.get ("udp")  
udp_dissector:call (buffer, pinfo, tree)
```

```
-- Send data to the UDP dissector's port 53 (DNS) handler
```

```
udp_table = DissectorTable.get ("udp.port")  
dnsdissector = udp_table.get_dissector (53)  
dnsdissector:call (buffer, pinfo, tree)
```

Other Methods

- Dumper
 - Used to dump data to files
- TextWindow
 - Creates a new window
- ProgDlg
 - Creates a progress bar dialog
- Address
 - Represents an address

Wireshark User Guide

- More information is available in the WSUG
 - <http://www.wireshark.org/docs/>
 - 10.4. Wireshark's Lua API Reference Manual

Summary

- We have created a dissector using
 - Proto
 - ProtoField
 - Tvb / TvbRange
 - TreeItem
 - Pref
 - DissectorTable
- We also provide Listeners and ability to create a post-dissector

Q & A

Questions?