SHARKFEST '13

Wireshark Developer and User Conference

Wireshark Dissectors – 3 ways to eat bytes.

Graham Bloice – Software Developer

Introduction

- Software Developer with Trihedral UK Limited
  - Use C++ and scripting for SCADA toolkit VTScada™
  - Use Wireshark with industrial tele-control protocols
- Wireshark Core Developer
  - First contributed to Wireshark in 1999
  - Maintain DNP3 dissector
  - Frequent contributor to “Ask Wireshark”
    - Mostly fixing formatting and converting “answers” to comments ☺

Topics to be Covered

- Wireshark internals brief overview
  - Where dissectors fit in
- Dissectors
  - Brief overview
  - Paths to implementation
  - Complexity and performance tradeoffs

Wireshark Internals

- Wireshark provides a framework for loading, dissection and visualization of network traffic
- Wireshark framework allows individual dissectors access to network data via libwiretap
- Wireshark framework provides utility functions for dissectors when dissecting data
- Wireshark framework allows dissectors to write out products of dissection

Dissectors overview

- Dissectors “register” their interest in data from a lower level protocol dissector, e.g. tcp port 54321
- The lower level dissector hands the payload body to the registered dissector
- Dissectors “pick apart” a protocol into the individual elements of the protocol message
- Each element of a protocol may have a type, e.g. integer, string, bit field, timestamp
- Dissectors provide elements that may be used in display filters

Dissector output

- Set the protocol column
- Set the info column
- Create tree entries as required
  - Create subtree entries for protocol components
  - Add values, text to tree entries
- Call sub-dissectors as required

Dissector Construction Options

- Text based
  - Built-in, with compilation – ASN.1, IDL
  - External, without compilation – Wireshark Generic Dissector (WSGD)*
- Scripting language based
  - Built-in, Lua*, Python (not in Windows)
  - External, Python (pyreshark)
- C based*
  - Traditional format, requires a development environment

Demonstration protocol

- Made up for this presentation
- Has a header with a byte indicating the message type, and a short (2 bytes in big-endian format) for the total message length (including the header)
- Commands are:
  - connect (20) and connect_ack (21) that have a long (4 bytes in little-endian format) id
  - disconnect (60) and disconnect_ack (61) that have a long (4 bytes in little-endian format) id
  - request_data and request_reply that have a byte indicating the data type and for the reply an actual data value;
    - Data type 0 – a little-endian short
    - Data type 1 – a little-endian long
    - Data type 2 – a 15 character string

Text based Dissectors

- Protocol definition held in text file(s) in human readable form
- Definitions are interpreted or compiled to produce a dissector
- Low barrier to entry, although ASN.1 and IDL requires a development environment to compile resulting dissector
- Interpreted text files are least performant option
- Least flexible option for access to libwireshark infrastructure

Wireshark Generic Dissector (WSGD)

- A Wireshark add-on created by Olivier Aveline, info at http://wsgd.free.fr/
- Allows dissection of a protocol based on a text description of the protocol elements
- Available for Windows and Linux as a plug-in module
- Has some limitations but relatively simple to use and doesn’t require a development environment

WSGD Dissectors

- Copy the appropriate version of the plugin into your Wireshark installation:
  - Global plugins
  - Personal plugins
- Copy (or create) the definition files in the required location:
  - As specified by the environment variable “WIRESHARK_GENERIC_DISSECTOR_DIR”
  - Profiles directory
  - User data directory
  - Global plugin directory
  - Wireshark main directory

WSGD Basics – Protocol Definition

```
# Protocol identification
PROTONAME               SharkFest 13 Protocol (WSGD)
PROTOSHORTNAME          SF13
PROTOABBREV             sf13

# Protocol parent, controls when dissector is called
PARENT_SUBFIELD         tcp.port
PARENT_SUBFIELD_VALUES  54321

# Message header, protocol starts with a header
MSG_HEADER_TYPE         T_msg_header

# Message type identifier, must be part of header
MSG_ID_FIELD_NAME       Function

# Message title field, shown in Info column
MSG_TITLE               InfoString

# Message size field, from field in header
MSG_TOTAL_LENGTH        Length

# Message body type:
MSG_MAIN_TYPE           T_msg_switch(Function)

# Field definitions
PROTO_TYPE_DEFINITIONS
include  sf13.fdesc;
```

WSGD Basics – Field Definitions I

```
# Message type enumeration
enum8 T_Type_message
{
  connect                20    # must be an integer
  connect_ack             -    # "-" indicates previous value + 1
  request_data           40
  request_reply           -
  disconnect             60
  disconnect_ack          -
}

# Header definition
struct T_msg_header
{
  byte_order             big_endian;

  T_Type_message         Function;
  uint16                 Length;
  hide var string        InfoString = print ("%s", Function);  # Note type conversion

  byte_order             little_endian;
}
```

WSGD Basics – Field Definitions II

```
# Messages
struct  T_msg_connect
{
  T_msg_header    Header;
  uint32          ID;
}

struct  T_msg_connect_ack
{
  T_msg_header    Header;
  uint32          ID;
}

struct  T_msg_disconnect
{
  T_msg_header    Header;
  uint32          ID;
}

struct  T_msg_disconnect_ack
{
  T_msg_header    Header;
  uint32          ID;
}
```

WSGD Basics – Field Definitions III

```
# Data value enumeration
enum8 T_Type_dataid
{
  read_short  0
  read_long   1
  read_string 2
}

struct T_msg_request_data
{
  T_msg_header          Header;
  T_Type_dataid         Data_ID;
}

struct T_msg_request_reply
{
  T_msg_header          Header;
  T_Type_dataid         Data_ID;
  switch(Data_ID)
  {
    case T_Type_dataid::read_short  : uint16 Data_Short;
    case T_Type_dataid::read_long   : uint32 Data_Long;
    case T_Type_dataid::read_string : string(15) Data_String;
  }
}

struct  T_msg_unknown
{
  T_msg_header          Header;
  raw(*)                end_of_msg;
}
```

WSGD Basics – Field Definitions IV

```
# Main switch
switch  T_msg_switch    T_Type_message
{
  case  T_Type_message::connect        : T_msg_connect         "";
  case  T_Type_message::connect_ack    : T_msg_connect_ack     "";
  case  T_Type_message::request_data   : T_msg_request_data    "";
  case  T_Type_message::request_reply  : T_msg_request_reply   "";
  case  T_Type_message::disconnect     : T_msg_disconnect      "";
  case  T_Type_message::disconnect_ack : T_msg_disconnect_ack  "";

  default : T_msg_unknown "";
}
```

Scripting language based dissectors

- Protocol definition held in text file(s) using the script language syntax
- Definitions are interpreted by the scripting language run-time
- Scripting run-time exposes access to libwireshark infrastructure
- Not all libwireshark infrastructure exposed
- No development environment required
- Faster than text dissectors, slower than C

Lua dissectors

- Lua is built-in to Wireshark (on most platforms)
- The lua support can be used to build dissectors, post-dissectors and taps
- init.lua in the global configuration directory is run at Wireshark start-up
- If disable_lua is not set to 0 runs init.lua from the personal configuration directory
- Loads all lua scripts (*.lua) in the global and personal plugins directory
- Runs any scripts passed on the command line with –X lua_script:xxx.lua

Lua dissector Basics – Protocol definition

```
-- declare the protocol
sf13_proto = Proto("sf13", "SharkFest'13 Protocol (lua)")

-- declare the value strings
local vs_funcs = {
    [20] = "connect",
    [21] = "connect_ack",
    [40] = "request_data",
    [41] = "request_reply",
    [60] = "disconnect",
    [61] = "disconnect_ack"
}

local vs_dataid = {
    [0] = "read short",
    [1] = "read long",
    [2] = "read string"
}
```

Lua dissector Basics – Field definition

```
-- declare the fields
local f_func = ProtoField.uint8("sf13.func", "Function", base.DEC, vs_funcs)
local f_len = ProtoField.uint16("sf13.len", "Length", base.DEC)
local f_id = ProtoField.uint32("sf13.id", "ID", base.DEC)
local f_dataid = ProtoField.uint8("sf13.data.id", "Data ID", base.DEC, vs_dataid)
local f_data_short = ProtoField.int16("sf13.data.short", "Data Short", base.DEC)
local f_data_long = ProtoField.int32("sf13.data.long", "Data Long", base.DEC)
local f_data_string = ProtoField.string("sf13.data.string", "Data String")

sf13_proto.fields = { f_func, f_len, f_id, f_dataid,
                      f_data_short, f_data_long, f_data_string }
```

Lua dissector Basics – Dissector function I

```
function sf13_proto.dissector(buffer, pinfo, tree)

    -- Set the protocol column
    pinfo.cols['protocol'] = "SF13"

    -- create the SF13 protocol tree item
    local t_sf13 = tree:add(sf13_proto, buffer())
    local offset = 0

    -- Add the header tree item and populate it
    local t_hdr = t_sf13:add(buffer(offset, 3), "Header")
    local func_code = buffer(offset, 1):uint()
    t_hdr:add(f_func, func_code)
    t_hdr:add(f_len, buffer(offset + 1, 2))
    offset = offset + 3

    -- Set the info column to the name of the function
    pinfo.cols['info'] = vs_funcs[func_code]

    -- dissect common part for connect and disconnect functions
    if func_code == 20 or func_code == 21 or func_code == 60 or func_code == 61 then
      -- A connect or connect ack or disconnect or disconnect_ack
      t_sf13:add_le(f_id, buffer(offset, 4))
    end
```

Lua dissector Basics – Dissector function II

```
    -- A request_data or request_reply message
    if func_code == 40 or func_code == 41 then
      -- dissect common part of request functions
      t_sf13:add(f_dataid, buffer(offset, 1))

      -- dissect request_reply data body
      if func_code == 41 then
        -- A request_reply
        local dataid = buffer(offset, 1):uint()
        offset = offset + 1
        if dataid == 0 then
          -- a read short data item
          t_sf13:add_le(f_data_short, buffer(offset, 2))
        end
        if dataid == 1 then
          -- a read long data item
          t_sf13:add_le(f_data_long, buffer(offset, 4))
        end
        if dataid == 2 then
          -- a read string data item
          t_sf13:add(f_data_string, buffer(offset, 15))
        end
      end
    end
end
```

Lua dissector Basics – Dissector registration

```
-- load the tcp port table
tcp_table = DissectorTable.get("tcp.port")

-- register the protocol to port 54321
tcp_table:add(54321, sf13_proto)
```

C based dissectors

- Protocol definition, implied in dissector source
- Dissector source file compiled into libwireshark or as a plugin
- All libwireshark infrastructure available
- Full development environment required
- High knowledge barrier to initial implementation
- Use existing dissectors as a guide
- Developers Guide is essential reading, also README.developer

C dissector installation

- Place source file in epan\dissectors
- Add entry for source to Makefile.common, epan \CMakeLists.txt
- Quick test compile in the dissectors directory, e.g.
  `nmake -f Makefile.nmake packet-sf13.obj`
- Rebuild Wireshark in the normal way

C dissector – preliminary declarations

```c
#include "config.h"
#include <glib.h>
#include <epan/packet.h>

#define SF13_PORT        54321
#define FUNC_CONNECT     20
#define FUNC_CONN_ACK    21
#define FUNC_REQ_DATA    40
#define FUNC_REQ_REPLY   41
#define FUNC_DISCONNECT  60
#define FUNC_DISC_ACK    61

/* A sample #define of the minimum length (in bytes) of the protocol data.
 * If data is received with fewer than this many bytes it is rejected by
 * the current dissector. */
#define SF13_MIN_LENGTH 4

static const value_string sf13_func_vals[] = {
  { FUNC_CONNECT,    "connect" },
  { FUNC_CONN_ACK,   "connect_ack" },
  { FUNC_REQ_DATA,   "request_data" },
  { FUNC_REQ_REPLY,  "request_reply" },
  { FUNC_DISCONNECT, "disconnect" },
  { FUNC_DISC_ACK,   "disconnect_ack" },
  { 0, NULL }
};

#define READ_SHORT       0
#define READ_LONG        1
#define READ_STRING      2

static const value_string sf13_data_type_vals[] = {
  { READ_SHORT,  "read short" },
  { READ_LONG,   "read long" },
  { READ_STRING, "read string" },
  { 0, NULL }
};

/* Initialize the protocol and registered fields */
static int proto_SF13 = -1;
static int hf_SF13_Func_Code = -1;
static int hf_SF13_Length = -1;
static int hf_SF13_ID = -1;
static int hf_SF13_Data_ID = -1;
static int hf_SF13_Data_Short = -1;
static int hf_SF13_Data_Long = -1;
static int hf_SF13_Data_String = -1;

/* Initialize the subtree pointers */
static gint ett_SF13 = -1;
static gint ett_SF13_hdr = -1;
```

C dissector – dissection function I

```c
/* Code to actually dissect the packets */
static int
dissect_SF13(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    /* Set up structures needed to add the protocol subtree and manage it */
    proto_item *ti, *hdr_ti;
    proto_tree *SF13_tree, *SF13_hdr_tree;
    /* Other misc. local variables. */
    guint offset = 0;
    guint8 func_code;
    guint8 data_id;

    /* Check that there's enough data */
    if (tvb_length(tvb) < SF13_MIN_LENGTH)
        return 0;

    /* Fetch some values from the packet header using tvb_get_*(). If these
     * values are not valid/possible in your protocol then return 0 to give
     * some other dissector a chance to dissect it.
     */
    func_code = tvb_get_guint8(tvb, offset);
    if (try_val_to_str(func_code, sf13_func_vals) == NULL)
        return 0;

    /*** COLUMN DATA ***/

    /* Set the Protocol column to the constant string of sf13 */
    col_set_str(pinfo->cinfo, COL_PROTOCOL, "SF13");
    col_clear(pinfo->cinfo, COL_INFO);
    col_add_str(pinfo->cinfo, COL_INFO,
                val_to_str(func_code, sf13_func_vals, "Unknown function (%d)"));
```

C dissector – dissection function II

```
/*** PROTOCOL TREE ***/

/* create display subtree for the protocol */
ti = proto_tree_add_item(tree, proto_SF13, tvb, 0, -1, ENC_NA);
SF13_tree = proto_item_add_subtree(ti, ett_SF13);

/* create subtree for the header */
hdr_ti = proto_tree_add_text(SF13_tree, tvb, 0, 3, "Header");
SF13_hdr_tree = proto_item_add_subtree(hdr_ti, ett_SF13_hdr);

/* Add an item to the subtree, see section 1.6 of README.developer for more
 * information. */
proto_tree_add_item(SF13_hdr_tree, hf_SF13_Func_Code, tvb, offset, 1, ENC_LITTLE_ENDIAN);
offset += 1;

/* Continue adding tree items to process the packet here... */
proto_tree_add_item(SF13_hdr_tree, hf_SF13_Length, tvb, offset, 2, ENC_BIG_ENDIAN);
offset += 2;

/* Now add items depending on the specific function code */
switch (func_code)
{
    case FUNC_CONNECT:
    case FUNC_CONN_ACK:
    case FUNC_DISCONNECT:
    case FUNC_DISC_ACK:
        proto_tree_add_item(SF13_tree, hf_SF13_ID, tvb, offset, 4, ENC_LITTLE_ENDIAN);
        offset += 1;
        break;
```

C dissector – dissection function III

```
case FUNC_REQ_DATA:
        case FUNC_REQ_REPLY:
            /* Dissect the common portion of the two functions */
            data_id = tvb_get_guint8(tvb, offset);
            proto_tree_add_item(SF13_tree, hf_SF13_Data_ID, tvb, offset, 1, ENC_LITTLE_ENDIAN);
            offset += 1;

            if (func_code == FUNC_REQ_REPLY) {
                switch (data_id)
                {
                    case READ_SHORT:
                        proto_tree_add_item(SF13_tree, hf_SF13_Data_Short, tvb, offset, 2, ENC_LITTLE_ENDIAN);
                        offset += 2;
                        break;
                    case READ_LONG:
                        proto_tree_add_item(SF13_tree, hf_SF13_Data_Long, tvb, offset, 4, ENC_LITTLE_ENDIAN);
                        offset += 4;
                        break;
                    case READ_STRING:
                        proto_tree_add_item(SF13_tree, hf_SF13_Data_String, tvb, offset, 15, ENC_LITTLE_ENDIAN);
                        offset += 15;
                        break;
                    default:
                        break;
                }
            }
            break;
        default:
            break;
    }

    /* Return the amount of data this dissector was able to dissect (which may
     * or may not be the entire packet as we return here). */
    return offset;
}
```

C dissector – protocol registration I

```
void
proto_register_SF13(void)
{
    /* Setup list of header fields  See Section 1.6.1 of README.developer for
     * details. */
    static hf_register_info hf[] = {
        { &hf_SF13_Func_Code,
            { "Function", "sf13.func",
              FT_UINT8, BASE_DEC, VALS(sf13_func_vals), 0x0,
              "Message Function Code Identifier", HFILL }
        },

        { &hf_SF13_Length,
            { "Length", "sf13.len",
              FT_UINT16, BASE_DEC, NULL, 0x0,
              "Message Length", HFILL }
        },

        { &hf_SF13_ID,
            { "ID", "sf13.id",
              FT_UINT32, BASE_DEC, NULL, 0x0,
              "Connection ID", HFILL }
        },

        { &hf_SF13_Data_ID,
            { "Data_ID", "sf13.data.id",
              FT_UINT8, BASE_DEC, VALS(sf13_data_type_vals), 0x0,
              "Data Type Identifier", HFILL }
        },

        { &hf_SF13_Data_Short,
            { "data_short", "sf13.data.short",
              FT_UINT16, BASE_DEC, NULL, 0x0,
              "Data Short", HFILL }
        },

        { &hf_SF13_Data_Long,
            { "data_long", "sf13.data.long",
              FT_UINT32, BASE_DEC, NULL, 0x0,
              "Data Long", HFILL }
        },

        { &hf_SF13_Data_String,
            { "data_string", "sf13.data.string",
              FT_STRING, BASE_NONE, NULL, 0x0,
              "Data String", HFILL }
        },
    };
```

C dissector – protocol registration II

```
/* Setup protocol subtree array */
    static gint *ett[] = {
        &ett_SF13,
        &ett_SF13_hdr
    };

    /* Register the protocol name and description */
    proto_SF13 = proto_register_protocol("SharkFest'13 Protocol (C)", "SF13", "sf13");

    /* Required function calls to register the header fields and subtrees */
    proto_register_field_array(proto_SF13, hf, array_length(hf));
    proto_register_subtree_array(ett, array_length(ett));
}


/* Simpler form of proto_reg_handoff_SF13 which can be used if there are
 * no prefs-dependent registration function calls. */
void
proto_reg_handoff_SF13(void)
{
    dissector_handle_t SF13_handle;

    /* Use new_create_dissector_handle() to indicate that dissect_SF13()
     * returns the number of bytes it dissected (or 0 if it thinks the packet
     * does not belong to SF13).
     */
    SF13_handle = new_create_dissector_handle(dissect_SF13, proto_SF13);
    dissector_add_uint("tcp.port", SF13_PORT, SF13_handle);
}
```

Comparison of dissectors

- For this simple “protocol”, Wireshark displays are almost identical
- WSGD and Lua dissectors are quick and easy to develop; edit file and restart Wireshark
- WSGD facilities are most limited option, Lua more advanced
- C dissectors easy to distribute, build an installer package, easy to contribute back to Wireshark
- WSGD 124 lines, Lua 89 lines, C 270 lines

Comparison of dissectors performance

- No visible difference for a small capture file, 11 packets
- Medium capture file (~ 1M packets) results:
  - WSGD: 2:16
  - Lua: 0:39
  - C: 0:20

Questions?

- Other dissector options?
- Future directions?