Writing Your Own Wireshark Packet Dissectors (ADVANCED)
March 31, 2008

Guy Harris

SHARKFEST '08
Foothill College
March 31 - April 2, 2008
Advanced dissector writing

Techniques needed for protocols that are “complicated”:

• Fragment reassembly
• Decryption and decompression
• Conversations and per-packet data
• Request/response matching
• Cleaning up allocated data
• Expert info
• Taps
“Simple” protocols

Many protocols are “simple”
• One PDU per bearer protocol PDU
• No encryption, compression, etc.
• Each PDU can be interpreted independently
“Complicated” protocols

But not all protocols are simple
• Fragmentation and reassembly
• PDU body encoding
• PDUs that require context from previous PDUs
• etc.
“Data sources”

Not all data in a dissected PDU comes directly from current lower-level PDU

- Can come from other lower-level PDUs (reassembly)
- Can come from decompressed/decrypted/etc. data
“Data sources” (cont’d)

Solution: named data sources

- Name, used as tab name in hex dump pane
- “Real data” tvbuff, containing data
  - Created with tvb_new_real_data
- Data source created with add_new_data_source
  - Added to list of data sources for packet
  - add_new_data_source(pinfo, tvbuff, name)
Dissector might need to do initialization/cleanup work when capture is opened

- Initialize internal data structures
- Free data left over from previous capture

(register_init_routine()) registers callback - callback has no arguments and no return value
Some higher-layer PDUs are built from multiple lower-layer PDUs

- IP fragmentation, IEEE 802.11 fragmentation, etc.
- Need to reassemble
- Lower-layer PDUs contain header plus payload
- Payloads of fragments are reassembled into higher-layer PDU
- Header of fragment indicates where it appears in higher-layer PDU
Some protocols identify fragments with PDU ID and byte offsets

- Examples: IPv4, IPv6
- ID identifies reassembled PDU (e.g., IP ID)
- Byte offset is offset within reassembled PDU of first byte of payload
- Last fragment is specially marked
Process payload

\texttt{fragment\_add\_check()} does "heavy lifting" of reassembly

- The first time this packet is seen:
  - Just returns NULL if fragment cut short by snaplen
  - Adds to reassembly based on \texttt{pinfo->src}, \texttt{pinfo->dst}, \texttt{id}
  - If all fragments found, saves as finished reassembly and returns \texttt{fragment\_data *} for finished reassembly
  - Otherwise, returns NULL

- All times after that, looks for finished reassembly and returns \texttt{fragment\_data *} for finished reassembly
Process payload (cont’d)

fragment_add_check() uses two GHashTable structures to keep track of fragments and reassemblies

• Initialize in dissector initialization routine
• Initialize first with fragment_table_init()
• Initialize second with reassembled_table_init()
Dissect reassembled payload

process_reassembled_data() does the “heavy lifting” to process a possibly reassembled PDU

• Checks whether reassembly done (head != NULL)
• If so, checks whether there’s more than one fragment
• If more than one fragment:
  • Creates new tvbuff for reassembled data
  • Adds a data source for it, with specified name
  • Adds items with information about fragments to protocol tree
• If only one fragment, creates subset tvbuff for payload
• If reassembly not done, returns NULL
Dissect reassembled payload (cont’d)

process_reassembled_data() needs ett_ and hf_ variables for subtrees and items it adds

• fragment_items structure specifies the variables

• Contains pointers to the variables
Reassembly with sequence # and offset

Some protocols identify fragments with PDU ID and sequence number

• Examples: TDS, TIPC
• ID identifies reassembled PDU
• Sequence number is ordinal number of fragment
  • 0-based or 1-based
• Last fragment specially marked
Process payload

Similar to fragment byte offsets

- `fragment_add_seq_check()`
- Takes sequence # rather than byte offset as argument
- Sequence # is 0-based
- For protocols with 1-based sequence #’s, subtract 1
Dissect reassembled payload

Same as for fragment byte offsets

- `process_reassembled_data()` hides the differences
Decryption/decompression

Some protocols encrypt or compress data in the PDU
• Must decrypt or decompress before processing
• Cannot decrypt/decompress in place
• Must generate new data array and tvbuff for data array
• Make a data source from tvbuff
Decrypt/decompress into new buffer

Allocate a buffer and decrypt/decompress into it:

```c
uint8  *buff;
tvbuff_t *new tvb;
int actual_size, captured_size;

actual_size = amount of decrypted/decompressed data;
captured_size = amount of that data we can generate from captured data;
buf = g_malloc(captured_size);
decrypt/decompress into buff;
```
Set up new tvbuff for buffer

Allocate a new tvbuff for the buffer:

```c
new_tvb = tvb_new_real_data(buff, captured_size, actual_size);
```

Arrange that the buffer be freed when the tvbuff is freed:

```c
tvb_set_free_cb(new_tvb, g_free);
```
Set up new data source

Arrange that the new tvbuff be cleaned up when the original tvbuff is cleaned up:

```
tvb_set_child_real_data_tvbuff(tvb, new_tvb);
```

Add the new tvbuff as a data source, so it shows up as a tab in the hex dump pane:

```
add_new_data_source(pinfo, new_tvb, name);
```

Do dissection with the new tvbuff
Conversations

Mechanism for keeping track of “flows” (connections, etc.)

- Identified by endpoint addresses and port numbers
- Addresses are address structures
  - Address type and value as bytes
- Ports are numbers
  - Conversation has “port type” identifying type of port
  - TCP, UDP, SCTP, IPX (socket), etc.
Conversation state

Information about flow attached to conversation

• Can set dissector for conversation
  • Protocol like SDP can indicate “Set up TCP session with this protocol”
  • Setting dissector means future packets will be dissected properly

• Can attach data to a conversation
  • Contains state information needed to dissect packets in conversation
Creating conversations

Use the function `conversation_new()`

- Caller must check for existing conversation first

- Arguments:
  - Frame number of first frame
  - Endpoint addresses
  - Port type and endpoint ports
  - Options

- Returns `conversation_t * handle`
Finding conversations

Use the function `find_conversation()`

• Arguments:
  • Frame number of first frame
  • Endpoint addresses
  • Port type and endpoint ports
  • Options

• Returns `conversation_t * handle`
Wildcards

conversation_new() options allow “wildcarding” of addresses and ports

- Can create conversation with one “wildcard” address and/or port
  - For UDP traffic where reply can come from address or port different from request destination
  - For “future” conversation where both endpoints are not known yet
- Can specify NO_ADDR2 and/or NO_PORT2
Wildcard matching

`find_conversation()` options control “wildcarding” of address and port

- Either fully specified or wildcard conversations can match
- Match with fewest wildcards wins
- Can specify `NO_ADDR_B` and/or `NO_PORT_B`
- Wildcard address or port can match *either* endpoint
Completing wildcard conversations

Matching can cause wildcards to be filled in

• Filling in is used for “future” connection-oriented conversations

• Protocol such as SDP can indicate “Set up TCP session with this protocol”

• Perhaps only one endpoint is known
  • Receiver of SDP message will connect to that endpoint
  • Full endpoint from which it connects is unknown

• Once connection is made, wildcarded endpoint is filled in by find_conversation()
Conversation dissector

Set with `conversation_set_dissector()`

• Takes `conversation_t *` and dissector handle as arguments

• Works for TCP, UDP, Datagram Congestion Control Protocol (DCCP), AppleTalk Transaction Protocol

• Takes precedence over heuristics and port matches
Conversation data

Each protocol can attach data to a conversation

• Data is opaque - not interpreted by conversation code
• Allocating and freeing is the dissector’s responsibility
• Check whether data already exists, then add data if it does not exist
• Data can be changed as you dissect packets in the conversation
  • Data should be changed only on first pass through packets - pinfo->fd->flags.visited false
Adding conversation data

Use the function `conversation_add_proto_data()`

- Arguments:
  - `conversation_t * handle`
  - Protocol number for protocol
  - `void *` pointing to data
- No return value
Use the function `conversation_get_proto_data()`

- **Arguments:**
  - `conversation_t * handle`
  - Protocol number for protocol

- **Returns** `void *` pointing to data, or `NULL` if no data for protocol
Per-packet data

Data needed in order to dissect a particular packet correctly

• Might come from previous packets
• Might come from per-conversation data updated by previous packets
Adding per-packet data

Use the function `p_add_proto_data()`

- **Arguments:**
  - `frame_data * handle`
  - Protocol number for protocol
  - `void *` pointing to data
- **No return value**
- **Check whether already present before adding**
  - Data should be added only on first pass through packets
Finding per-packet data

Use the function `p_get_proto_data()`

• Arguments:
  • `frame_data * handle`
  • Protocol number for protocol

• Returns `void *` pointing to data, or `NULL` if no data for protocol
Request/response matching

Many protocols are request-response protocols

- Decoding response might require info from request
- User might want response to show frame # of request
- User might want time between request and response
Request/response matching table

If multiple requests in flight, protocol probably has request ID field

- Use GHashTable or se_tree with request ID as key
  - One table per conversation, not one global table!
- Store relevant information as value
  - Information needed to dissect response (e.g., request type)
  - Time stamp of request
  - Frame #s of request and response (0 means unknown)
Freeing allocated data

Data allocated privately by dissectors eventually needs to be freed

- If `se_` allocators used, freeing happens automatically
- Otherwise, need to free data in your initialization routine
Expert analysis

Log of “possibly interesting” behavior in a capture

- Allows users to get a summary of what they might want to look at

- Four severity levels:
  - Chat - interesting events in normal traffic flow, such as TCP SYN
  - Note - notable but not unusual events, such as HTTP 404
  - Warn - unusual events, such as a connection failure
  - Error - serious problem, such as a malformed packet
Expert information groups

The general type of condition an item describes:

- Bad checksum
- Protocol sequence problem (discontinuous sequence numbers, retransmissions, etc.)
- Error response (gives error code)
- Request code (typically at Chat level)
- Undecoded
- Reassembly problem
- Malformed packet
Setting expert info

“Expert info” is a property of a protocol tree item
• Must have an item to which expert info is attached
• Added with `expert_add_info_format()`:  
  • `packet_info * (pinfo)` pointer  
  • `proto_item *` pointer to item  
  • group  
  • severity  
  • `printf`-style format string and arguments
Taps and tap listeners

Mechanism for producing statistics, etc., from packets

- Dissectors provide taps
- Statistics routines provide tap listeners
- Tap listeners attach to taps
Taps can supply pre-digested data to listeners

- Register tap by name in `proto_register_` routine
- Queue packet for tap in dissection routine
  - Pass (non-auto!) data structure with pre-digested data, if tap supplies any
Tap listeners

Tap listeners process data supplied by taps

• Per-packet “packet” callback arguments are:
  • pinfo
  • Dissection information (including protocol tree)
  • Pre-digested information from dissector, if any

• “Display” callback called when accumulated information should be displayed or updated

• Tap listeners with UI cannot be shared between Wireshark and TShark - with an exception...
stats_tree taps

stats_tree tap does the UI work for you

- Displays a tree view
  - A list view is just a tree view with one level
- All you do is add nodes and update nodes
- Works with both Wireshark and TShark
- stats_tree tap can be a plugin (see stats_tree plugin)
Further information

• Everything Gerald mentioned in “Further information” in the previous session
  • (Except for “Next session”; no infinite looping here :-))
• doc/README.request_response_tracking
• doc/README.tapping
• doc/README.stats_tree
Bonus material

Sample code
if (doing defragmentation && this is part of fragmented PDU) {
    head = fragment_add_check(tvb, payload_offset, pinfo, ID,
        fragment_table, reassembled_table, byte_offset,
        fragment_data_len, true_for_last_fragment);
    next_tvb = process_reassembled_data(tvb, offset, pinfo,
        "name", head, &frag_items, &update_col_info, tree);
} else {
    if (this is not part of fragmented PDU || this is the first fragment) {
        next_tvb = tvb_new_subset(tvb, payload_offset, -1, -1);
        if (part of fragmented PDU)
            pinfo->fragmented = TRUE;
        else
            pinfo->fragmented = FALSE;
    } else
        next_tvb = NULL;
}
Do dissection after reassembly

```c
if (next tvb == NULL) {
    /* Just show this as a fragment. */
    if (check_col(pinfo->cinfo, COL_INFO)) {
        col_add_fstr(pinfo->cinfo, COL_INFO, something to show this as a fragment);
    }
    if (head && head->reassembled_in != pinfo->fd->num) {
        if (check_col(pinfo->cinfo, COL_INFO)) {
            col_append_fstr(pinfo->cinfo, COL_INFO,
                            " [Reassembled in %#u]", head->reassembled_in);
        }
    }
call_dissector(data_handle, tvb_new_subset(tvb,
            payload_offset, -1, -1), pinfo, parent_tree);
pinfo->fragmented = save_fragmented;
} else {
    hand next tvb to the next dissector;
}
```