Design Plan - Codec

James Viner

Project Summary

Codec is a pair of programs, a decoder and an encoder. The decoder takes in a .pcap file and, if it is determined to be one with a zerg data payload, prints the message within in a human-readable format. The encoder takes in the formatted output from running the decoder and turns it back into a .pcap file.

Features Targeted

Generate Checksums

Allow functionality of properly generating checksums when using the encoder. This doesn't require any additional complexity to be added to the decoder, given that this field will not be stored or displayed to the user, but the encoder will need to calculate both the IPv4 and UDP checksums when the packet is re-encoded. Both protocols use some form of error-checking in the form of 1's complement addition of the header fields which shouldn't be too difficult to implement and then place into the relevant checksum fields.

Big-Endian .pcap Support

Allow the reading and writing of .pcap headers in Big-Endian format. For the decoder, the magic number field at the top of the .pcap header indicates whether the header is in Big/Little-Endian based on how it reads out relative to one's own computer. In the case that it reads out backwards on our VMs, that means the packet is in Big-Endian format and requires additional calls to ntohl() or ntohs() in order to have its values parsed correctly. For the encoder, when instructed to with a -b option, the opposite functions htonl() or htons() should be called to convert data into network byte order.

GPS-output with Special Symbols

Make the GPS output print degrees/minutes/seconds with appropriate symbols. I may be missing some element of complexity, but it does seem like this just requires ensuring fields are lined up properly on the back-end and printing using the specific formatting instructions from the rubric. The encoder needs to be able to strip these additional symbols when recreating the original GPS data and ensure it is packed back into the proper order.

Architecture

Data

Structs are going to need to be created for each individual packet header (.pcap, IPv4, UDP) as well as each of the zerg data packet types. An overarching struct will be used to hold a disassembled version of a given packet in the form of struct pointers, as well as a pointer to a struct for every type of zerg data payload. The correct pointer will be placed into the relevant spot during the runtime of the load_zerg_data() function, while the rest will be set to NULL, with the print statements later in the program printing contents based on each data payload type if the pointer is not NULL.

Significant Functions - Decoder

`int load_headers(struct packet *current_packet)`

Loads the topmost portion of a given packet into the relevant fields of the packet struct. This function will do the processing of the portion of the packet that is least expected to vary between given sample files, throwing out any packets that are truncated or otherwise deemed to be malformed. Returns an integer 0 for failure and 1 for success for the purposes of either passing the current packet along for further processing or acquiring a new one.

`int load_zerg_data(struct packet *current_packet)`

Loads the zerg header and data payload for a given packet into the relevant fields of the packet struct. This function does the heavy-lifting in regards to the handling of the four different zerg packet payload types, ensuring the data within is not malformed, and filling in a pointer to the created struct into the correct field in the struct packet instance for later processing. Similar to the above function, returns 0 for failure and 1 for success.

`void print_packet_contents(struct packet *current_packet)`

Calls one of four helper print functions each designed to print a specific type of zerg packet. This central printing function handles just the determining of which printing helper function needs to be called in order to properly display a given packet's contents based on which packet data pointer is not set to NULL.

Significant Functions - Encoder

```
void calculate_packet_length(struct packet *current_packet)
```

Uses logic based on the zerg payload type to calculate packet lengths for the .pcap, IPv4, and UDP headers. May require an additional argument for length of string for the zerg message payload type.

```
void generate_headers(struct packet *current_packet)
```

Loads the static sections of the .pcap, Ethernet, IPv4, and UDP headers into the packet struct.

```
int parse_packet_contents(struct packet *current_packet)
```

Calls one of four helper print functions each specifically designed to parse a specific type of zerg packet. This central parsing function handles just the determining of which parsing helper function needs to be called in order to properly load a given packet's contents based on what type of zerg data payload is received. Returns 0 if the packet was deemed to be malformed based on the read-in fields or 1 on success.

```
void generate_checksums(struct packet *current_packet)
```

Generates the checksums for the UDP and IPv4 headers based on the relevant information placed within the rest of the packet.

```
void write_output(struct packet *current_packet, char *file)
```

Writes the properly-formatted output as bytes to a file.

User Interface

Users should be able to give the decoder a given file for decoding and receive a human-readable print to the console of the contents of all zerg packets within. Similarly, users should be able to give the encoder this human-readable print and a filename and receive a .pcap file that can be decoded again to receive the same originally-decoded message.

General Approach

The general approach is to begin with the decoder, getting it to read binary data from .pcap files into structs first and then attempting to print the contents, all under ideal conditions and starting with a single packet. Then, the implementation of error-handling for malformed packets should be the next step, still only working with single packets. Finally, adding dynamic memory allocation for an array of struct packets that way data can be read to and from multiple packets in a single packet capture. Once complete functionality has been established, work will begin on the decoder.

Some work may need to be done to the output format of the decoder at this time to ensure that the encoder does not have to do an unnecessary amount of work to parse the output and load it back into a struct. Similar to before, I will be ensuring that the encoder is able to work on the output from a single packet first under ideal conditions, then slowly introduce complexity through error-handling and allowing multiple packets. Work on the Big Endian and checksum validation features will occur during the initial setup of the encoder, when it is still working with a single packet, that way it can be scaled up easily later.