REVERSING CLIENT AND SERVER

Emanuelle (Manny) Crespi
UID: ecrespi, 111556427   ENEE459B - 0102

Running the Client and Server:

The client is allowed to run after the server has been set up to wait for a connection. (Figure 1)

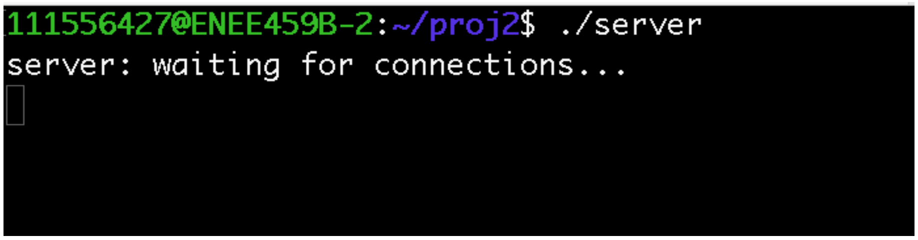

Figure 1: Server Application at Runtime

Following the connection, a prompt for the binary executable is displayed on the client application. It is important to give a valid IP as a command line argument to the client binary for full functionality (Figure 2).

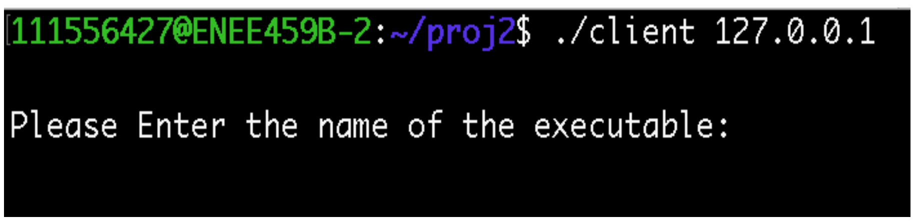

Figure 2: Client Application at Runtime

For the software to send a data packet, a valid binary must exist within the "../elfs" directory from where the binary is running. This allows the application to parse, verify and send the file. Without this, the client program will complain that there is no file to open before terminating. Below is an example for both a successful run (Figure 3) that sends a packet to the server, and an unsuccessful run (Figure 4) that terminates without a correct file name.

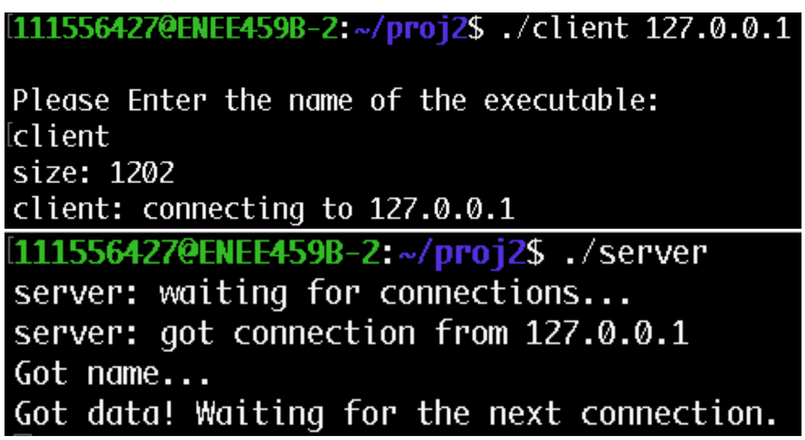

Figure 3: Successful Run Client-Server Interaction

```
111556427@ENEE459B-2:~/proj2$ ./client 127.0.0.1

Please Enter the name of the executable:
fasdf

File "../elfs/fasdf" NOT FOUND!
111556427@ENEE459B-2:~/proj2$ clear
```

Figure 4: Unsuccessful Run

There are also other cases to consider where the binary is not the correct format. Attempting to use another file type that is in the correct directory will cause the program to output "File is not ELF." before terminating (Figure 5).

```
111556427@ENEE459B-2:~/proj2$ ./client 127.0.0.1

Please Enter the name of the executable:
client1
File is not ELF.
111556427@ENEE459B-2:~/proj2$
```

Figure 5: Attempting to process an incorrect binary format

We can also verify a successful packet sending interaction by checking the directory where the server binary resides. The file name should be present on the server with the same name chosen on the client side prompt. The file contents are shown and described below for the example of a successful run above (Figure 6).

```
Name: client
0x1202
0x8048980
0x9
0x20
0x1f
0x28
2
```

<u>FORMAT – Head =Binary Name</u>

- size of text section (bytes)
- entry of text section
- # of program headers
- size of program headers
- # of section headers
- size of section headers
- data form

Figure 6: The contents/format of the file 'client1' after a successful data transfer

Parsing/Obfuscating/Sending Data packet:

The parsing of the files is correct in retrieving the attributes of the binary data. This has been verified by careful observation of the source code in IDA Pro, followed by comparison to the output of 'readelf –h –S'.

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8048980
  Start of program headers:          52 (bytes into file)
  Start of section headers:          11840 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         31
  Section header string table index: 28

Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        08048154 000154 000013 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            08048168 000168 000020 00   A  0   0  4
  [ 3] .note.gnu.build-i NOTE            08048188 000188 000024 00   A  0   0  4
  [ 4] .gnu.hash         GNU_HASH        080481ac 0001ac 000024 04   A  5   0  4
  [ 5] .dynsym           DYNSYM          080481d0 0001d0 000250 10   A  6   1  4
  [ 6] .dynstr           STRTAB          08048420 000420 00014d 00   A  0   0  1
  [ 7] .gnu.version      VERSYM          0804856e 00056e 00004a 02   A  5   0  2
  [ 8] .gnu.version_r    VERNEED         080485b8 0005b8 000050 00   A  6   1  4
  [ 9] .rel.dyn          REL             08048608 000608 000010 08   A  5   0  4
  [10] .rel.plt          REL             08048618 000618 000108 08  AI  5  24  4
  [11] .init             PROGBITS        08048720 000720 000023 00  AX  0   0  4
  [12] .plt              PROGBITS        08048750 000750 000220 04  AX  0   0 16
  [13] .plt.got          PROGBITS        08048970 000970 000008 00  AX  0   0  8
  [14] .text             PROGBITS        08048980 000980 001202 00  AX  0   0 16
  [15] .fini             PROGBITS        08049b84 001b84 000014 00  AX  0   0  4
```

Figure 7: Parsing of the ELF files are reliable

Following the verification of the ELF file header, the client application establishes an <u>application level handshake</u> with the server by sending the value 17 in a buffer. The server expects this value on the other end, and responds by sending 18 before moving on to the next phase (Figure 8). On the client side, the next two sends will complete the transaction. The first send will contain a <u>request value</u> appended to the name of the ELF file, and the last send will forward the entire packet.

The server is waiting to receive each of these transactions, so it expects to retrieve the value 21 from the buffer after receiving the first data packet.

NOTE: The ELF filename is also placed within the packet before sending. The entire packet is encrypted by some key generated with a combination of functions beforehand

CLIENT

```
add     esp, 10h
mov     [ebp+power_key_enc], eax
mov     byte ptr [ebp+packet.var1], 21
movzx   eax, [ebp+tcp_buf+1]
add     eax, 1          ; taking the char recieved from server in bu
mov     byte ptr [ebp+packet.var1+1], al
sub     esp, 4
push    92              ; n
lea     eax, [ebp+filename] ; copying filename to structure packet
add     eax, 8
push    eax             ; src
lea     eax, [ebp+packet]
add     eax, 3          ; offset 3 will be data section in packet
push    eax             ; dest
call    _memcpy         ; copy over the filename into head of data
add     esp, 16
```

SERVER

```
lea     eax, [ebp+packet]
mov     ecx, 256
mov     edi, edx
mov     esi, eax
rep movsd
mov     eax, esi
mov     edx, edi
movzx   ecx, word ptr [eax]
mov     [edx], cx
lea     edx, [edx+2]
lea     eax, [eax+2]
push    ebx
call    encrypt             ; encrypt before send
add     esp, 40Ch
lea     edx, [ebp+packet]
lea     eax, [ebp+var_1158]
mov     ecx, 100h
mov     edi, edx
mov     esi, eax
rep movsd
mov     eax, esi
mov     edx, edi
movzx   ecx, word ptr [eax]
mov     [edx], cx
lea     edx, [edx+2]
lea     eax, [eax+2]
push    0                   ; flags
push    1026                ; n
lea     eax, [ebp+packet]
push    eax                 ; buf
push    [ebp+fd]            ; fd
call    _send               ; send 1026 bytes of data (struct p
```

Figure 8: Prep work for first (encrypted) send of packet structure to server

A brief explanation of the structure is necessary at this point before discussing the formatting/decryption/encryption of the message that the client prepares and sends to the server. The structure is identified as a 1026 byte word before it is sent to the server, so the following identity is what I have used to identify packet sends/retrieval (Figure 9).

```
struct packet{
    short a;
    short b;
    char data[1024]
}
```

Figure 9: Packet structure on the client side for sending ELF attributes to server

Retrieval of the first encrypted packet is identified to pass on the server side, only if 21 is present within its decrypted format [packet.a == 21]. We can see below that the socket connection is terminated, if this condition is not met. When the condition is met, the server will proceed to wait for the next and final packet (Figure 10).

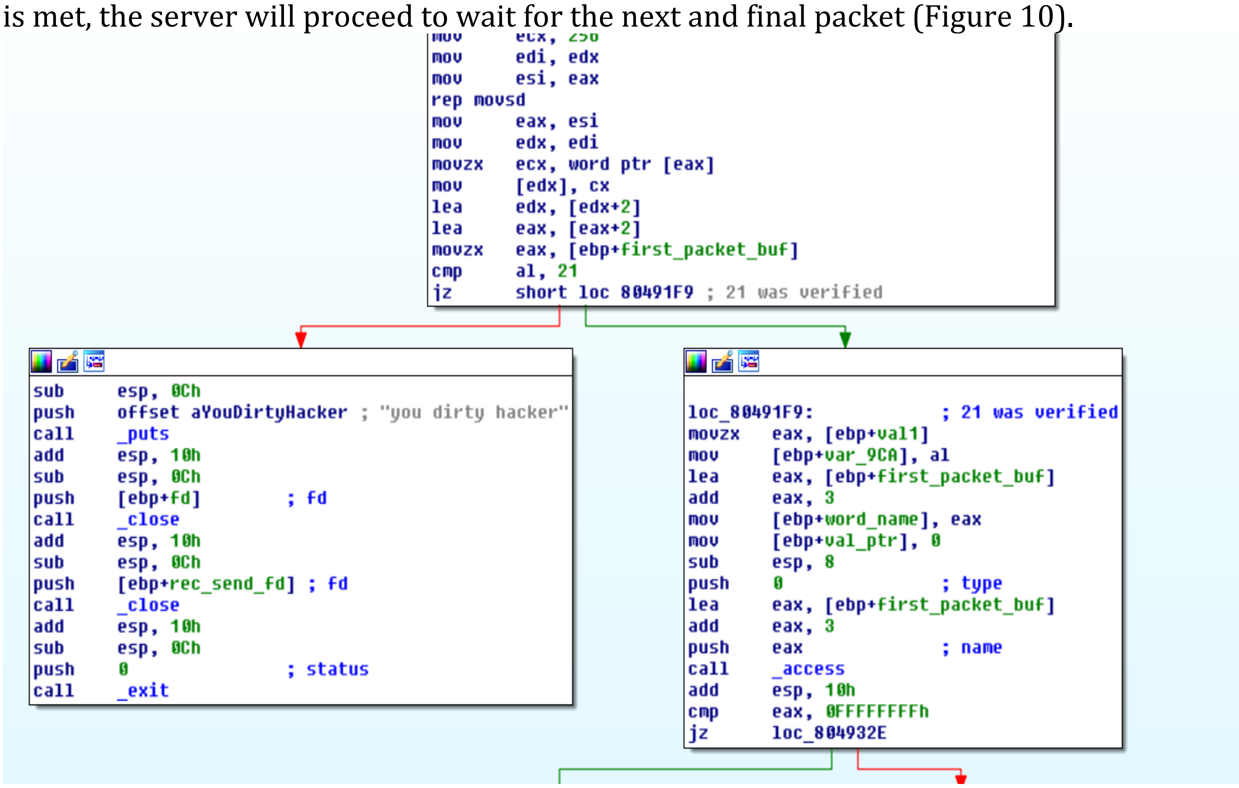

Figure 10: First packet is retrieved and verified by the server

There is one last request made by the client as it sends the final packet to the server. It checks for the same old request as before! Figure 11 below shows a comparison on the re-named variable 'old_request' that has not been touched since the last request. This is implying a comparison on 21 yet again.

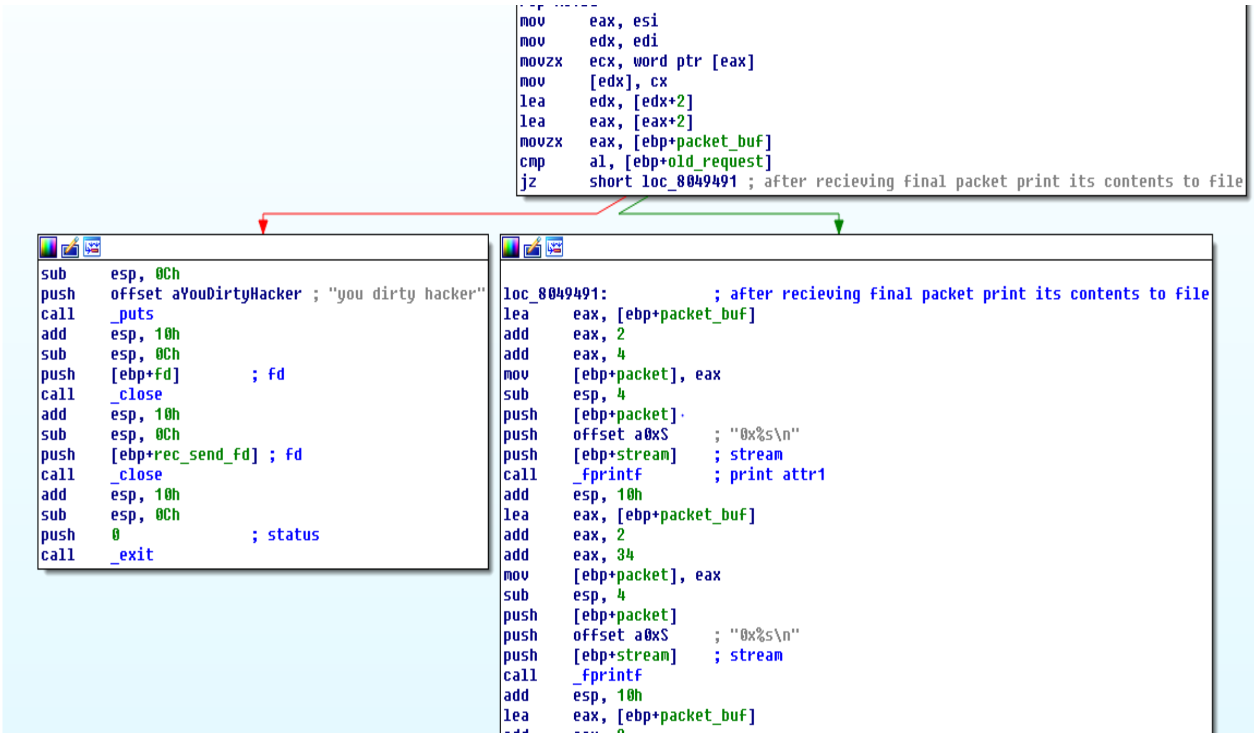

Figure 11: Passing the second request to process data from client and write to a file

Immediately after these two requests are fulfilled, the server proceeds to write the parsed attributes to the file. The offsets by which the server extracts from the structure indicate a mapping for the data elements that are written to the file since they are the same offsets as the ones used by the client to populate the buffer before sending the packet. Figure 12 below shows this overlap, and Figure 13 describes a mapping for where to find these 7 attributes in the buffer packet.

```
push    offset asc_8049C8F ; "%x"
lea     eax, [ebp+packet]
add     eax, 6
push    eax             ; s
call    _sprintf
add     esp, 10h
movzx   eax, [ebp+var_D2E]
sub     esp, 4
push    eax
push    offset asc_8049C8F ; "%x"
lea     eax, [ebp+packet]
add     eax, 0BAh
push    eax             ; s
call    _sprintf
add     esp, 10h
mov     eax, [ebp+var_CF8]
mov     eax, [eax+18h]
sub     esp, 4
push    eax
push    offset asc_8049C8F ; "%x"
lea     eax, [ebp+packet]
add     eax, 24h
push    eax             ; s
call    _sprintf
add     esp, 10h
mov     eax, [ebp+var_CF8]
movzx   eax, word ptr [eax+2Ch]
movzx   eax, ax
sub     esp, 4
push    eax
push    offset asc_8049C8F ; "%x"
lea     eax, [ebp+packet]
add     eax, 42h
push    eax             ; s
call    _sprintf
add     esp, 10h
mov     eax, [ebp+var_CF8]
movzx   eax, word ptr [eax+2Eh]
movzx   eax, ax
sub     esp, 4
push    eax
push    offset asc_8049C8F ; "%x"
lea     eax, [ebp+packet]
add     eax, 60h
push    eax             ; s
call    _sprintf
add     esp, 10h
mov     eax, [ebp+var_CF8]
movzx   eax, word ptr [eax+30h]
movzx   eax, ax
sub     esp, 4
push    eax
push    offset asc_8049C8F ; "%x"
lea     eax, [ebp+packet]
add     eax, 7Eh
push    eax             ; s
call    _sprintf
add     esp, 10h
mov     eax, [ebp+var_CF8]
movzx   eax, word ptr [eax+2Ah]
movzx   eax, ax
sub     esp, 4
push    eax
push    offset asc_8049C8F ; "%x"
```

CLIENT

```
push    [ebp+packet]
push    offset a0xS       ; "0x%s\n"
push    [ebp+stream]      ; stream
call    _fprintf          ; print attr1
add     esp, 10h
lea     eax, [ebp+packet_buf]
add     eax, 2
add     eax, 34
mov     [ebp+packet], eax
sub     esp, 4
push    [ebp+packet]
push    offset a0xS       ; "0x%s\n"
push    [ebp+stream]      ; stream
call    _fprintf
add     esp, 10h
lea     eax, [ebp+packet_buf]
add     eax, 2
add     eax, 64
mov     [ebp+packet], eax
sub     esp, 4
push    [ebp+packet]
push    offset a0xS       ; "0x%s\n"
push    [ebp+stream]      ; stream
call    _fprintf
add     esp, 10h
lea     eax, [ebp+packet_buf]
add     eax, 2
add     eax, 154
mov     [ebp+packet], eax
sub     esp, 4
push    [ebp+packet]
push    offset a0xS       ; "0x%s\n"
push    [ebp+stream]      ; stream
call    _fprintf
add     esp, 10h
lea     eax, [ebp+packet_buf]
add     eax, 2
add     eax, 124
mov     [ebp+packet], eax
sub     esp, 4
push    [ebp+packet]
push    offset a0xS       ; "0x%s\n"
push    [ebp+stream]      ; stream
call    _fprintf
add     esp, 10h
lea     eax, [ebp+packet_buf]
add     eax, 2
add     eax, 94
mov     [ebp+packet], eax
sub     esp, 4
push    [ebp+packet]
push    offset a0xS       ; "0x%s\n"
push    [ebp+stream]      ; stream
call    _fprintf
add     esp, 10h
lea     eax, [ebp+packet_buf]
add     eax, 2
add     eax, 184
mov     [ebp+packet], eax
sub     esp, 4
push    [ebp+packet]
push    offset aS         ; "%s\n"
push    [ebp+stream]      ; stream
call    _fprintf
```

SERVER

Figure 12: The offsets are useful for mapping the data within the packet

The mappings of the seven attributes are implied by the order in which the server writes to the file.

It is also important to see that there is enough space at the head of the data packet where at least 6 bytes of data are free.

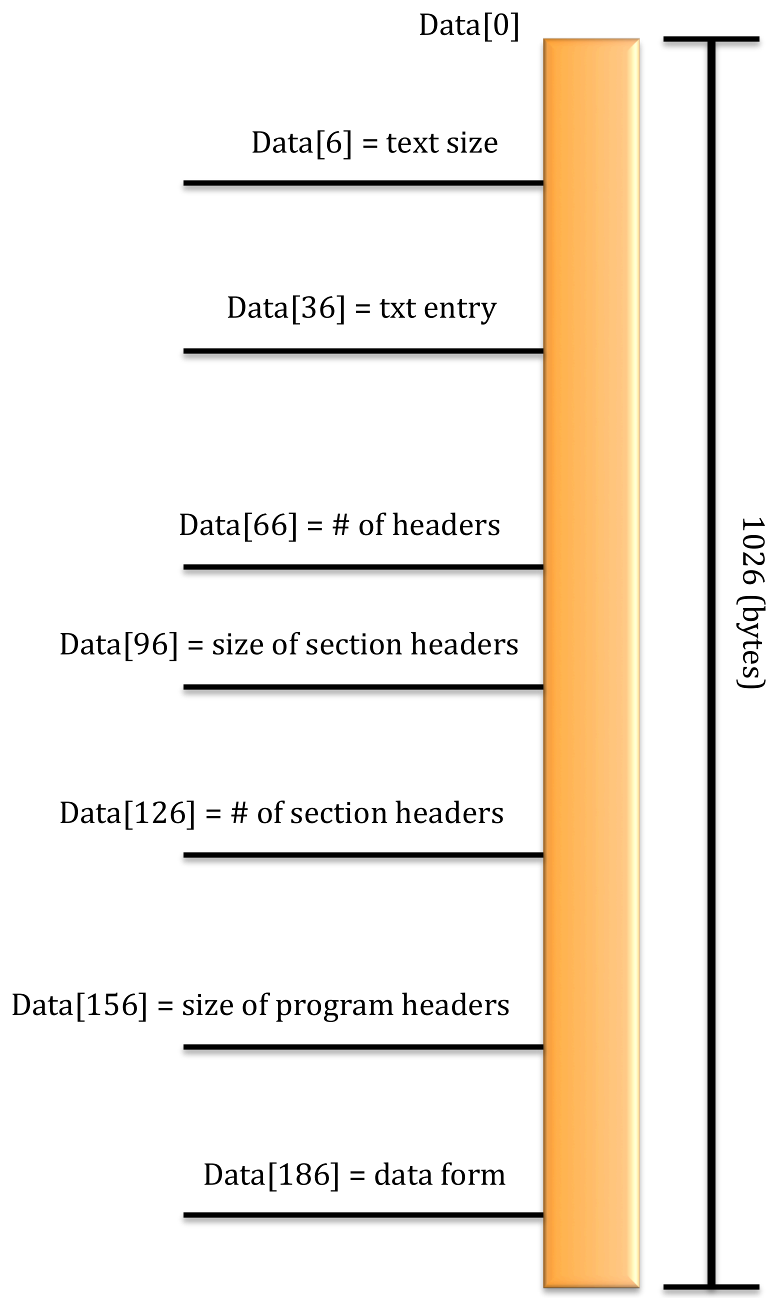

Figure 13: Mappings of the attributes residing in data

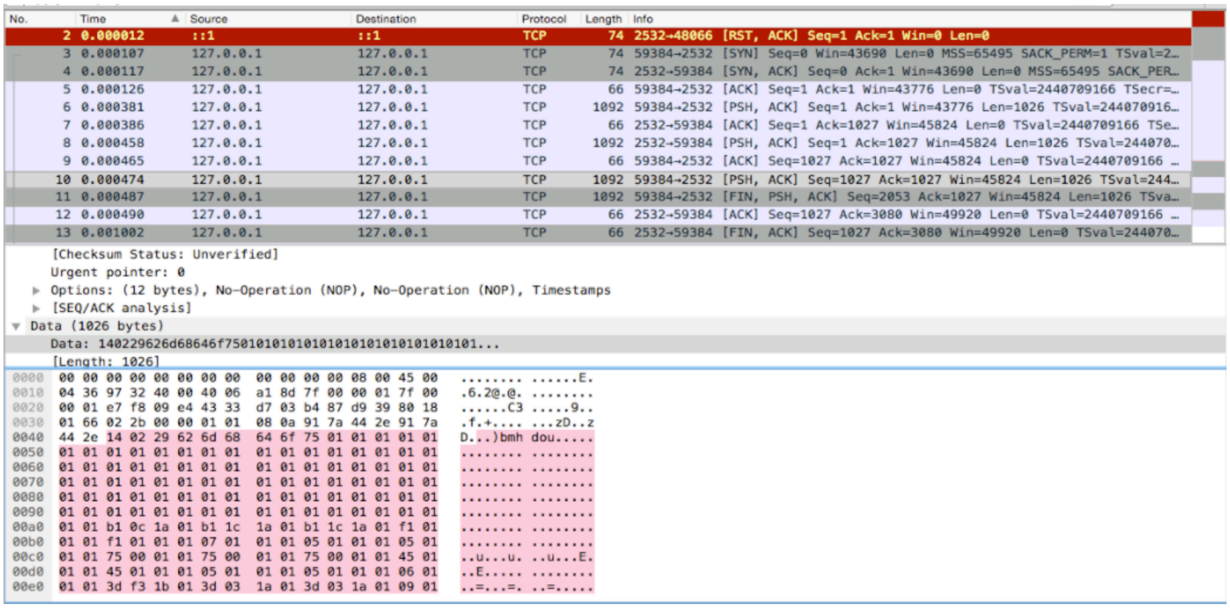

Figure 14: Intercepting client- server communication with Wireshark

At this point we know the order of the packets being sent and by whom. Using wire-shark to read intercepted messages, it has become clear that the message 09e44333 shows up in multiple packets during the sending process. Even though the encryption/decryption scheme is overly complicated to reverse, we can still analyze a consistent pattern between packets being sent to the server.

Vulnerabilities/Limitations:

It is also important to note that there is no checksum used on the other end to detect malformed data. If a packet, sent from client to server, has been intercepted and modified, the server would never be aware of this. It would continue to load the data into its database as long as there is a correct header = 21. Cannot find any problems with the obfuscation, encryption, decryption of the messages since I can hardly decipher it myself!

Logical Flow Chart (Server-Client Relationship):

Finally, with the data collected throughout this analysis of client and server, it would help to have a logical model (Figure 15) describing every interaction between these two applications. This way, any useful information I may have missed may be reconsidered as an added bonus to the reversing for this design.

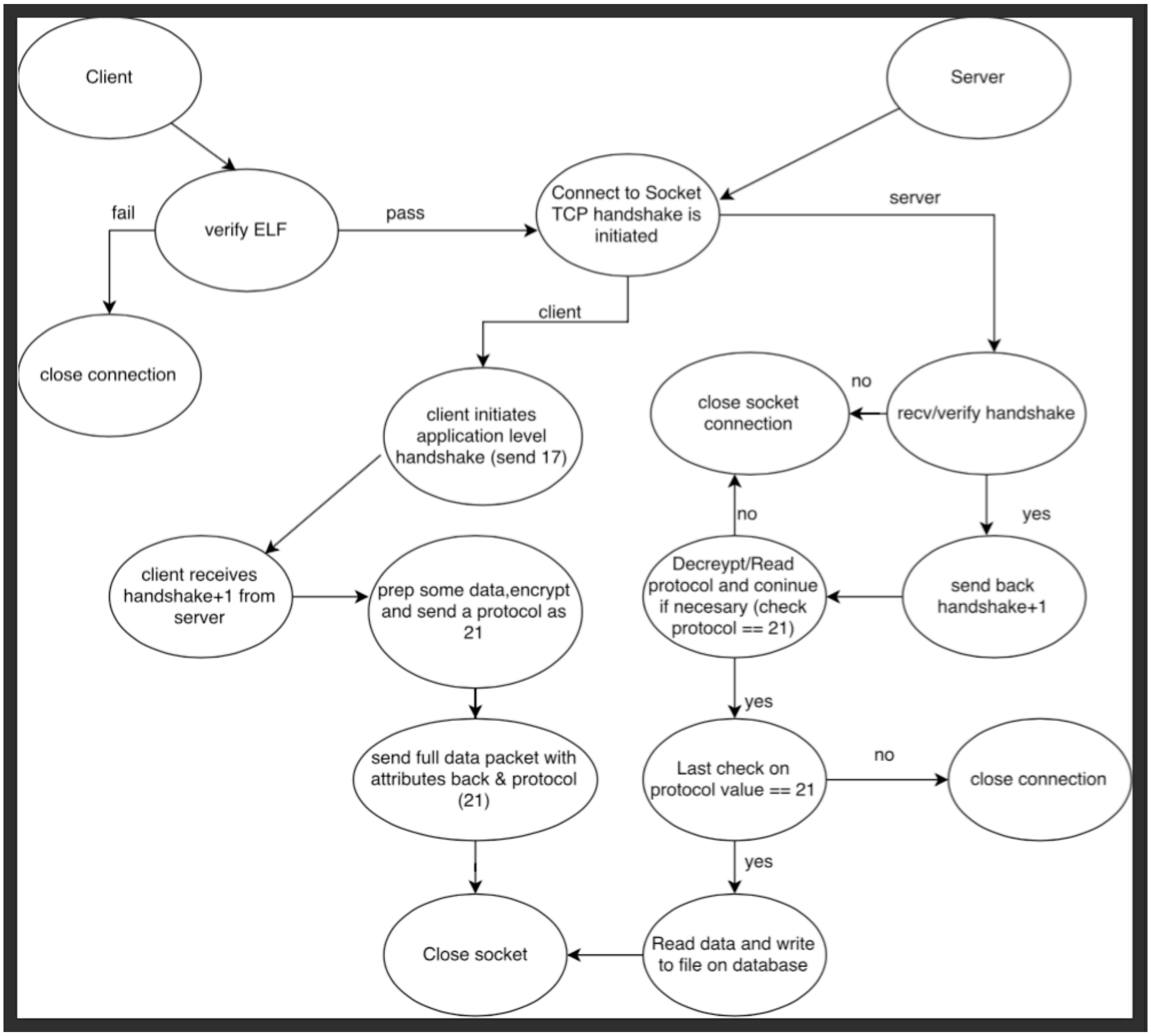

Figure 15: Logical Model Client-Server Flow Chart