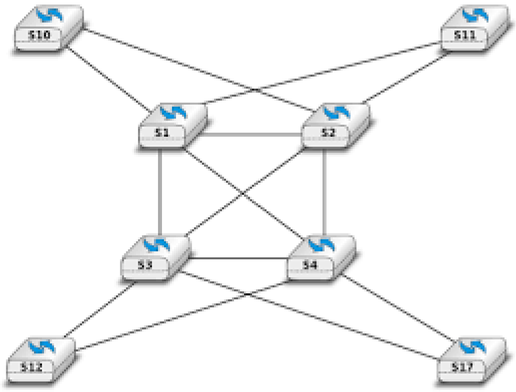
Dissecting Emotet’s network communication protocol • Raashid Bhat

Published: 2019-04-22 · Archived: 2026-04-05 18:47:17 UTC

April 22, 2019

Dissecting Emotet’s network communication protocol



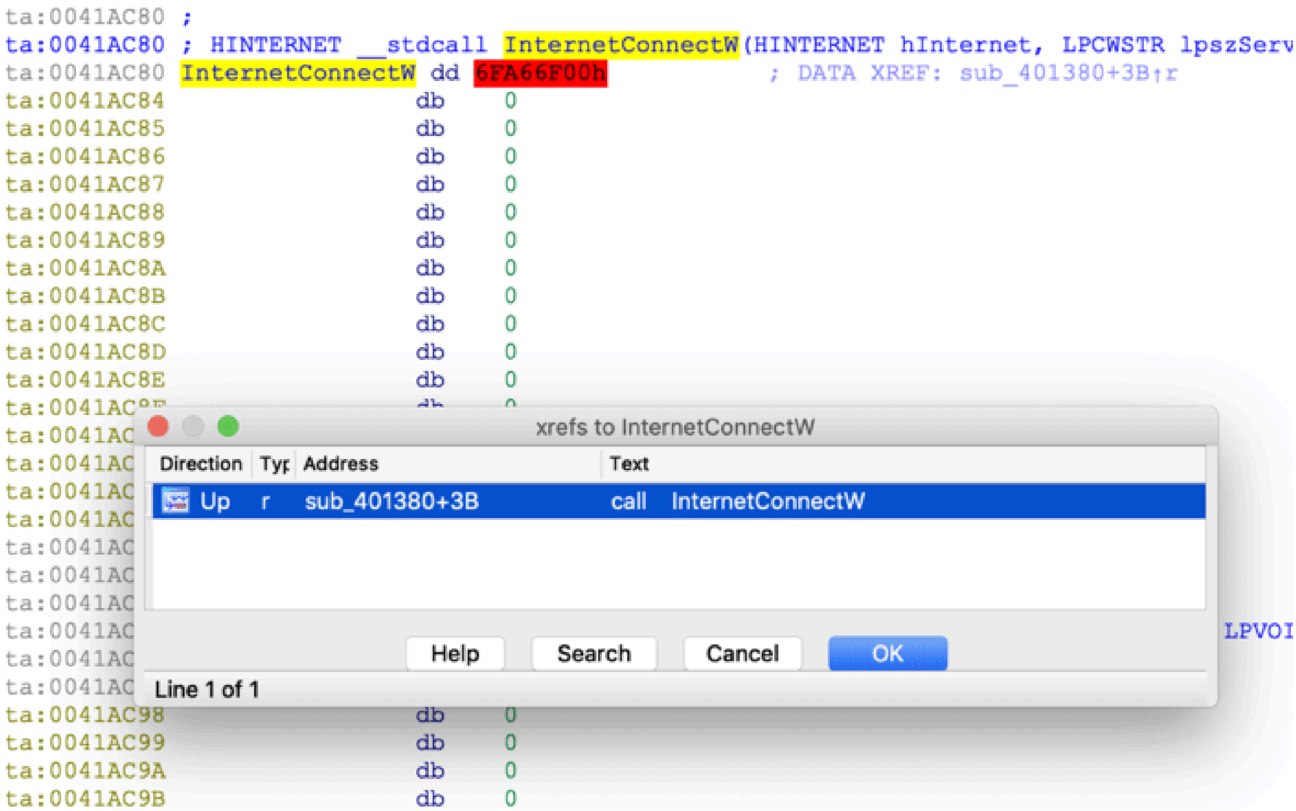
Request Packet format

Communication protocol for any malware lies at the core of its functionality . It is the essential way for any malware to communicate and receive further commands . Emotet has a complex communication format . Its peculiarities are the way the protocol is built and sent across the network . Knowing internal details of its communication format is essential to keep tabs on it . In this post we are going to analyze Emotet communication format .

we will be skipping the unpacking and reconstruction part , as it is irrelevant to this topic of discussion .

In this post , we will be specifically looking for areas of interest in the binary , there will be some parts that are analyzed preemptively .

An unpacked emotet sample has around ~100 functions , as populated by IDA . Going through each of them to look for communication subroutines would be “A short in the dark” . The easiest way would be to look for network API calls and xrefs would sort out most of the dirty work for us



Luckily in emotet., there is only one xref to this API call , which perhaps would be the subroutine where the communication to c2 server happens . This subroutine receives an encrypted and compressed packet with parameters like c2 server, port and sends it out . Xrefing back few subroutines would land us to the place where the packet is formulated . For comprehension , let's name this subroutine as ConnectAndSend

```
push    ebp
mov     ebp, esp
sub     esp, 190h
push    esi
push    edi
call    GetTickCount
xor     edx, edx
mov     [ebp+botID], offset aDesktopx6v45ii ; "DESKTOPX6V45IIN_761FD900"
mov     ecx, 0EA60h
div     ecx
mov     eax, dword_41AF50
push    offset aDesktopx6v45ii ; "DESKTOPX6V45IIN_761FD900"
mov     [ebp+Uptime], eax
lea     esi, [edx+0CD140h]
call    lstrlenA
mov     [ebp+BotIdLen], eax
lea     eax, [ebp+VersionInformation]
push    eax             ; lpVersionInformation
mov     [ebp+VersionInformation.dwOSVersionInfoSize], 11Ch
call    RtlGetVersion
lea     eax, [ebp+var_70]
push    eax
call    GetNativeSystemInfo
movzx   eax, [ebp+var_76]
imul    eax, 64h
add     eax, [ebp+VersionInformation.dwMajorVersion]
lea     ecx, [eax+eax*4]
mov     eax, [ebp+VersionInformation.dwMinorVersion]
lea     eax, [eax+ecx*2]
imul    ecx, eax, 64h
movzx   eax, [ebp+var_70]
add     ecx, eax
mov     eax, large fs:30h
mov     [ebp+OsVer], ecx
lea     ecx, [ebp+ProcList]
mov     eax, [eax+1D4h]
mov     [ebp+terminalSessionID], eax
mov     eax, Crc32
mov     [ebp+C3c32Hash], eax
call    sub_4022E0
```

Tracking back xfrefs , we finally reach to the subroutine where the packet is generated . And , based on API calls and variables used , we can easily name few local variables and subroutines used , for example Botid, crc32, etc

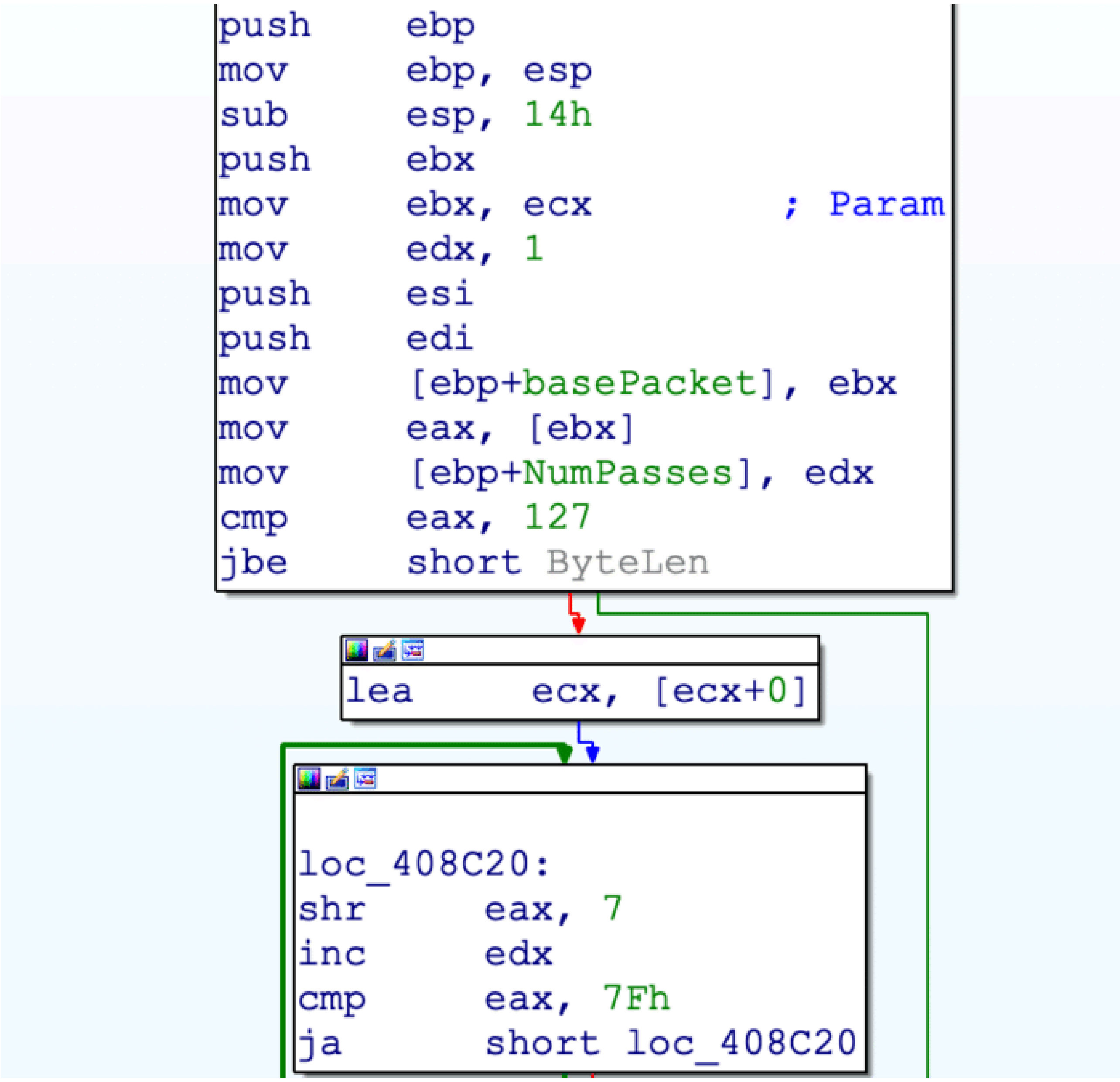
Based on how stack variable are set , we get an idea that a struct is formulated . The definition of the structure would be as following

```
 struct Emotet_BotInfo
{
    DWORD Uptime;
    BYTE *BotID;
    DWORD BotIDLen;
    DWORD MajMinOSversion;
    DWORD TermSessID;
    DWORD Crc32HashBinary;
    BYTE *ProcList;
    DWORD ProlistLen;
```

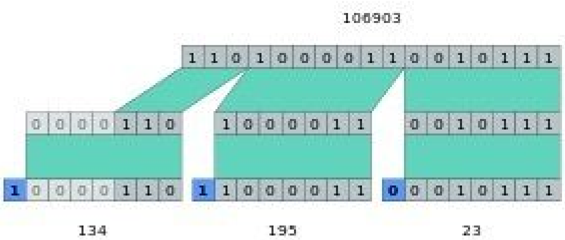
```
    DWORD PluginsInstalled[];
    DWORD PluginsLen;

};
```

Uptime - Measure of uptime of the infection
BotID - Botnet Identifier (unique per infection)
BotIDLen - Length of BotID
*MajMinOSversion *- Operating system identifier*
**TerminalSessID* - Terminal Session ID
Crc32HashBinary - CRC32 hash of binary
ProcList - List of running processes (comma segregated)
PluginsInstalled - Array of DWORD consisting of MODID's of plugins installed

This structure is passed on to a function that calculates total round size based on some bit shifts . This shifting gives us a clue about the format of the packet . Lets look at these patterns



Translating it to a code snippet would roughly be equivalent to




```
 towrite = number & x7f
number >>= 7
```

This code encodes an integer to LEB128 or Little Endian Base 128 format (VARINT). And one of the serialized buffer formats that support it is the google protobuf format , this clue again makes the reversing equation easy for us . Some old emotet analysis blogs support our assumption .

Emotet has two packets one being encapsulated in the other . The inner layer lets call it base packet. Base packet fundamentally is a group of entries with metadata information . Metadata includes type of data and an index number particular to the entry . Entries have a simple structure , but varies according to the type of entry

```
 Struct EmotetEntry
{
    VARINT ULEB128_EntryLength ;
    BYTE Data[ULEB128_EntryLength];

}
```

Emotet's base packet has three type of data entries, and are marked by numbers in the metadata

Type of element and type of data entry is specified in the metadata field

so, the complete definition of base packet would be something like this

```
 struct BaseEmotetPacket
{
    BYTE MetaData
    Struct EmotetEntry
    {
        VARINT ULEB128_EntryLength ;
        BYTE Data[ULEB128_EntryLength];

    }

}[n];
```

MetaData is a bitfield data type , which consists of
*** 0-3 bits - Type of data field ***
***3-7 bits - Index Number of Data field ***

Where *index* is a incremental number and type is an enum

```
 Enum Type
{
    Type 5 : Machine dependent endian WORD size integer
    Type 2 : Buffer Struct { VARINT ULEN128_Size, BYTE data[ULEN128_Size];
    Type 0 : ULEN128 encoded variant
 }
```

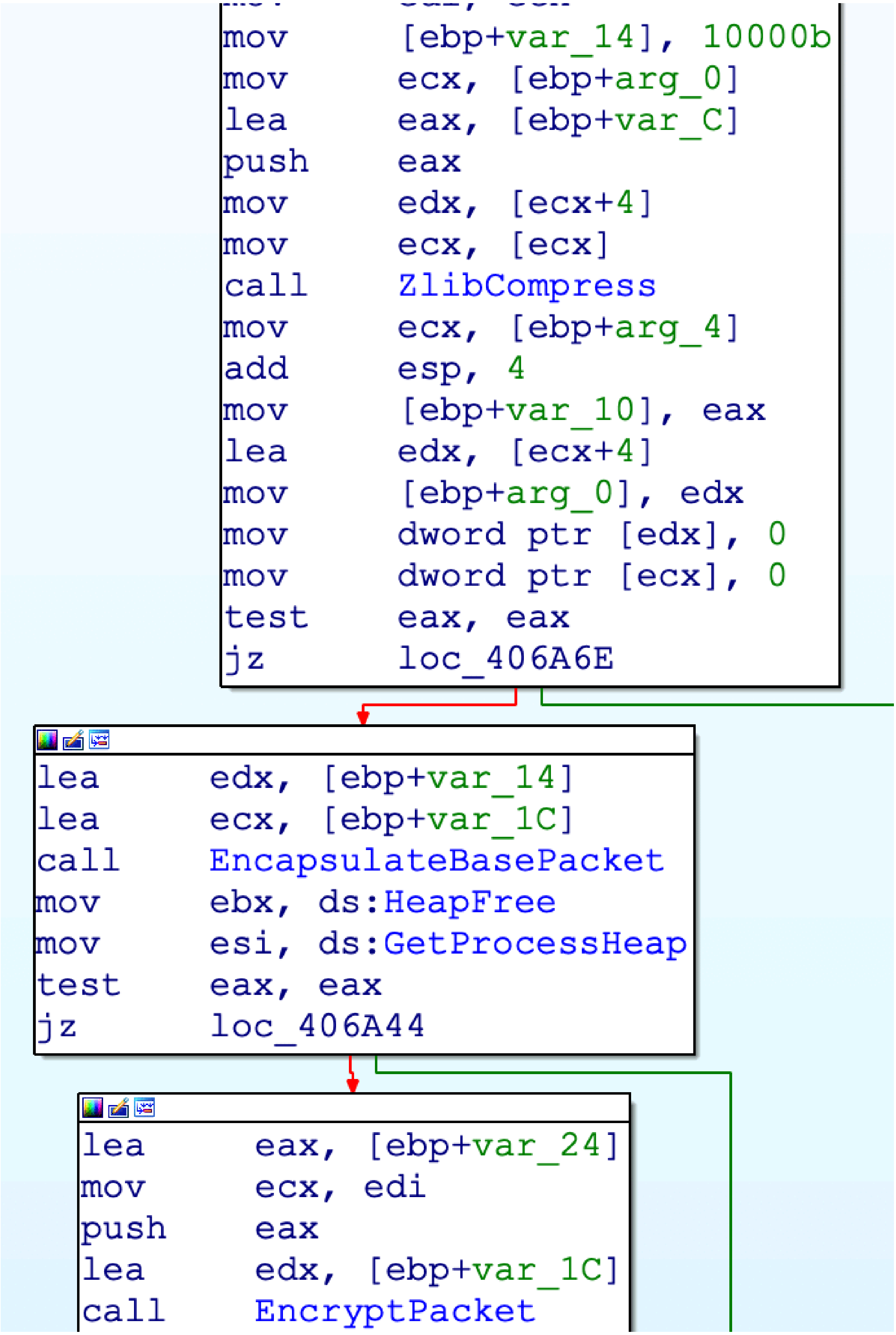
The code to add an entry in base packet can be defined in python as

```
def AppendElement(protoBuf, type, value, itemNum):
   protoBuf = protoBuf + struct.pack("B", ( (itemNum << 3) | type ) & 0xff)

   if type == 5: #DWORD Copy 32bit integer as it is
       return protoBuf + struct.pack("I", value)
   if type == 2: # Memory Buffer struct {VARINT ULEB128_Size, void * buf}
       return protoBuf + encode(len(value)) + value
   if type == 0: # encode DWORD in ULEB128
       return protoBuf + encode(value)
```

Later on , base packet is compressed and further more encapsulated in another packet

The definition of the final packet is almost the same as the base packet , but the only subtle difference is that it only has one field , which is the encapsulated base packet

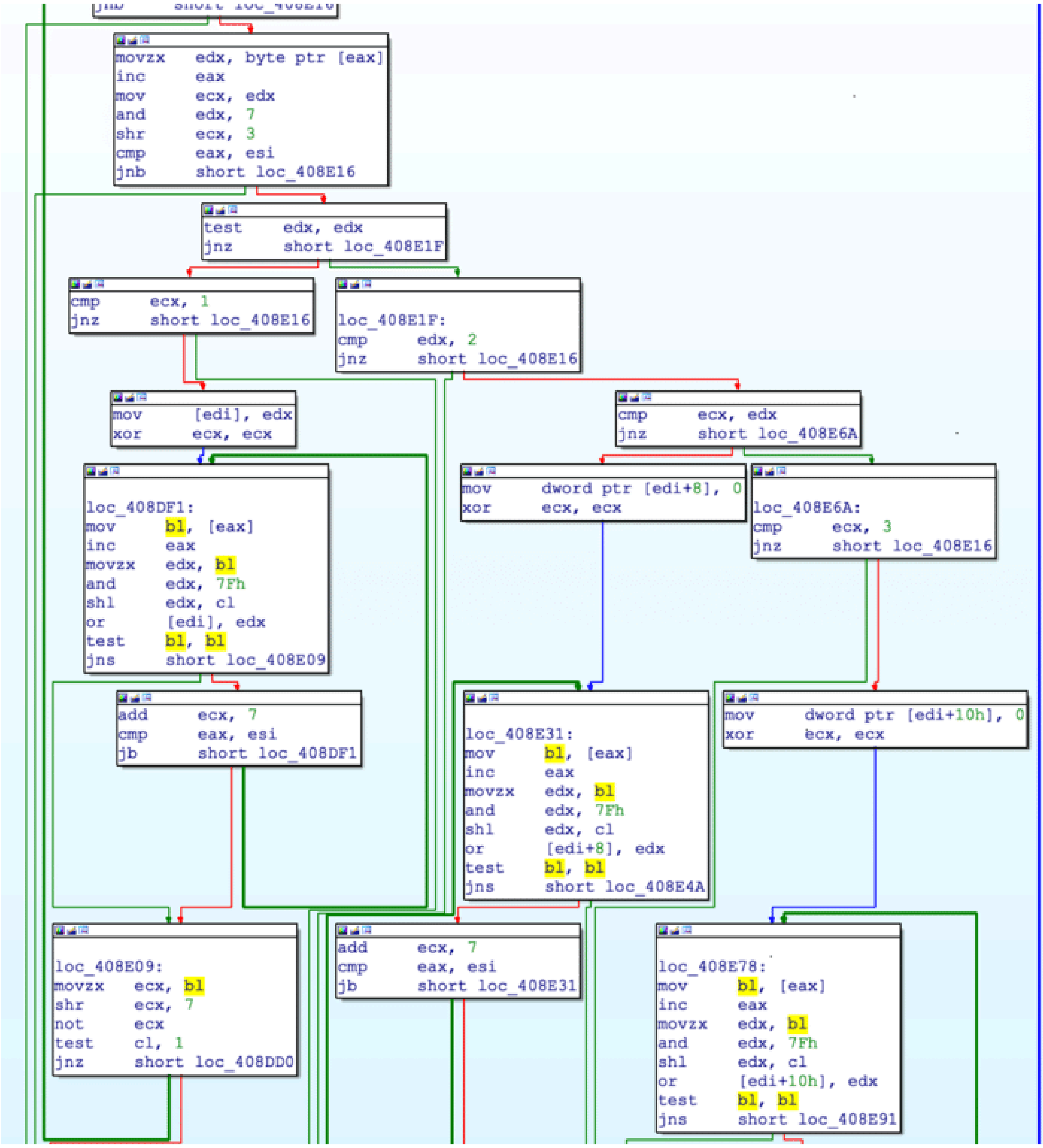
```
 struct FinalPacket
{
    BYTE MetaData;
    Struct BaseEmotetPacket BasePacket;



};
```


mov [ebp+var_14], 10000b
mov ecx, [ebp+arg_0]
lea eax, [ebp+var_C]
push eax
mov edx, [ecx+4]
mov ecx, [ecx]
call ZlibCompress
mov ecx, [ebp+arg_4]
add esp, 4
mov [ebp+var_10], eax
lea edx, [ecx+4]
mov [ebp+arg_0], edx
mov dword ptr [edx], 0
mov dword ptr [ecx], 0
test eax, eax
jz loc_406A6E
lea edx, [ebp+var_14]
lea ecx, [ebp+var_1C]
call EncapsulateBasePacket
mov ebx, ds:HeapFree
mov esi, ds:GetProcessHeap
test eax, eax
jz loc_406A44
lea eax, [ebp+var_24]
mov ecx, edi
push eax
lea edx, [ebp+var_1C]
call EncryptPacket

```
add     esp, 4
test    eax, eax
```

This data is sent to c2 server immediately after encrypting the final packet .

Response Packet format



Response data from c2 from received is decompressed , and the plain text data is supplied to a subroutine for deserialization .

The response data field uses the same variable length integer encoding and is almost structured in the same way .

Response format is complex and tentative for each type of request and bot configuration .

Similarly like base request packet, this structure consists of a type and number bitfield , which determines which type of data field is it . In case of response , it has three of them

*1 : Main module packet **
**2 : Binary update data*
3 : Deliverables data

```
 struct EmotetResponse
{
    unsigned char Number : 4;
    unsigned char Type : 4;
    unsigned char ModID; // Each module has modid ( 0 for main module)

    unsigned char Number : 4;
    unsigned char Type : 4;
    VARINT UpdateBinLen; // Varint Type ULEB128 Encoded
    BYTE BinaryBlob[UpdateBinLen]; // Update Binary PE FILE

    unsigned char Number : 4;
    unsigned char Type : 4;
    VARINT deliverablesLen;

        struct deliverables_
        {
            unsigned char Number : 4;
            unsigned char Type : 4;
            unsigned char ModID; // PluginModid

            unsigned char ExeFlag; // "" 3 - Plugin , 2 -  WriteElevatedExecute, 1 - writeExecute"""

            VARINT PluginLen; // Varint Type ULEB128 Encoded
            BYTE PluginBinaryBlob[PluginLen]; // Update Binary PE FILE

        }

}
```

32

Kudos

32

Kudos
